Natural Voice User Interface

I: Basics of Computer Voice Recognition
II: Subclassing Windows and Controls

Marek W. Zawolek

CA-World 2000
eBusiness Solutions in Internet Time
JP155SN & JP165SN

Abstract

This paper presents principles of voice communication with a computer using IBM's ViaVoice. It concentrates on explaining the basic problems of computer voice recognition and shows how to connect an application to the voice recognition engine through a class that wraps the functions of ViaVoice's SMAPI.DLL. Voice output is achieved through ViaVoice's text-to-speech Virtual Voices ActiveX control. A mechanism for handling events generated by voice input is added to the windows and controls, allowing for easy writing of completely voice-driven applications. Throughout the paper Jasmine *ii* and CA Visual Objects 2.5 are used for demonstration purposes, but the techniques presented apply to development with different programming languages and tools as well. As an example, a fully voice-driven version of Jasmine's CA Store (Fashion Boutique), with a CA-Visual Objects 2.5 front end, is shown.

Introduction

Natural languages are quite complex and have difficult grammars; they are full of exceptions, double meanings of words and similar pronunciations of words with different meanings. In spite of all these shortcomings, natural languages are by far the most popular form of human communication. So, for people who work with computers, it is a dream to communicate with their machines the way they communicate with other human beings; just say a word and the computer will react. How much easier would it be to say a thousand numbers, request their processing and hear the answer, than to sit in front of a computer, type a thousand figures, point and click and read the answer. So, how far are we from the computer technology of Star Trek?

This paper attempts to show that actually we are not too far. Already today, using an average PC equipped with a microphone, speakers, a good sound card and, of course, proper software, we can talk to our computers and hear them answering. The most important component from our point of view, appropriate software, is available from several companies; the best-known voice recognition software is produced by IBM and by Dragon. Throughout this paper IBM's ViaVoice will be used as a basis of voice communication.

It is one thing to say something and send it to a PC and another to write software that will make sense out of it. For instance, if I say the word “two,” I would like my software to enter “2” into a particular control instead of entering “2” or “too” or “to” just anywhere. So, as programmers, we have to use the capabilities of IBM’s ViaVoice to recognize words, but it is up to us to write software that controls anything else. For the task of doing “anything else,” I chose CA-Visual Objects because of its ability to communicate with DLLs written in C and with ActiveX controls, as well as with various databases. The most important thing for future development seems to be native access from CA-Visual Objects 2.5 to Jasmine. As I come from the CLIPPER community, it is also easier for me to write a large and well-performing application using CA-Visual Objects than any other language.

Connection Between CA-Visual Objects and IBM’s ViaVoice

Translation of spoken words into strings of characters is the basis of voice communication with a computer. After long research IBM came to the market with its family of software which can perform this task. Using vocabularies with pronunciation and spelling of words, improved by analysis of context, IBM’s software achieves good recognition. It comes with extra tools for further training and adding new words into dictionaries in order to improve recognition of a particular speaker and to understand special words or names. The latest version of IBM’s voice recognition software is called ViaVoice 98. Its main functions are controlling Windows by voice and dictating documents to a word processor. However, a computer language, like CA-Visual Objects, can communicate with the speech engine of ViaVoice via an API in order to extract spoken words as character strings.

Another part of ViaVoice 98 is the text-to-speech engine that does the opposite; it translates character strings to spoken words, completing a two-way voice communication. Third-party software can access this text-to-speech engine via ActiveX technology, so CA-Visual Objects serves well here, also.

Access to the Voice Recognition Engine

IBM provides with its ViaVoice an API called SMAPI.DLL. This DLL, written in C, ensures proper communication with the speech engine; it receives character strings for recognized words and it passes messages to the speech engine. Therefore, in order to communicate with ViaVoice, our CA-Visual Objects program must access functions of SMAPI.

Basic SMAPI Functions

There are plenty of functions that control all possible features of the IBM’s ViaVoice. The complete description can be found in the *SMAPI Reference* (1). The most important of these functions, together with their CA-Visual Objects prototypes, are:

SmOpen, **SmClose** – opens and closes the voice recognition session

```
_DLL FUNCTION SmOpen (iArgs AS SHORTINT, pSmArg AS PTR) AS SHORTINT;
   PASCAL:SMAPI.SmOpen
_DLL FUNCTION SmClose() AS INT PASCAL:SMAPI.SmClose
```

SmConnect, **SmDisconnect** – connects to and disconnects from the speech engine

```
_DLL FUNCTION SmConnect (iArgs AS SHORTINT, pSmArg AS PTR, pReply AS PTR) AS;
   SHORTINT PASCAL:SMAPI.SmConnect
_DLL FUNCTION SmDisconnect (iArgs AS SHORTINT, pSmArg AS PTR, pReply AS PTR) AS;
   INT PASCAL:SMAPI.SmDisconnect
```

SmDefineVocab, **SmEnableVocab**, **SmDisableVocab**, **SmUndefineVocab** – defines, enables, disables and "undefines" a vocabulary

```
_DLL FUNCTION SmDefineVocab (pszVocab AS PSZ, iWords AS SHORT,;
   ptrVocWords AS PTR, pReply AS PTR) AS SHORTINT PASCAL:SMAPI.SmDefineVocab
_DLL FUNCTION SmEnableVocab (pszVocab AS PSZ, pReply AS PTR) AS SHORTINT;
   PASCAL:SMAPI.SmEnableVocab
_DLL FUNCTION SmDisableVocab (pszVocab AS PSZ, pReply AS PTR) AS SHORTINT;
   PASCAL:SMAPI.SmDisableVocab
_DLL FUNCTION SmUndefineVocab (pszVocab AS PSZ, pReply AS PTR) AS SHORTINT;
   PASCAL:SMAPI.SmUndefineVocab
```

SmDefineGrammar – defines grammar that is used for recognition of entire phrases

```
_DLL FUNCTION SmDefineGrammar (pszVocab AS PSZ, pszFilaName AS PSZ,;
   liOptions AS LONG, pReply AS PTR) AS SHORTINT PASCAL:SMAPI.SmDefineGrammar
```

SmRecognizeNextWord – enables the recognition of the next word

```
_DLL FUNCTION SmRecognizeNextWord (pReply AS PTR) AS SHORTINT;
   PASCAL:SMAPI.SmRecognizeNextWord
```

SmGetFirmWords, **SmGetInfirmWords** – gets firmly recognized words and best guesses

```
_DLL FUNCTION SmGetFirmWords (Reply AS PTR, pWords AS DWORD PTR,;
   ptrWords AS PTR) AS SHORTINT PASCAL:SMAPI.SmGetFirmWords
_DLL FUNCTION SmGetInfirmWords (Reply AS PTR, pWords AS DWORD PTR,;
   ptrWords AS PTR) AS SHORTINT PASCAL:SMAPI.SmGetInfirmWords
```

SmMicOn, **SmMicOff** – switches the microphone on and off

```
_DLL FUNCTION SmMicOn (pReply AS PTR) AS SHORTINT PASCAL:SMAPI.SmMicOn
_DLL FUNCTION SmMicOff (pReply AS PTR) AS SHORTINT PASCAL:SMAPI.SmMicOff
```

Wrapping SMAPI Functions by a CA-Visual Objects Class

It is not too convenient to use SMAPI functions directly. More friendly to CA-Visual Objects programmers is wrapping all SMAPI functions in a class that will allow communication, but hide unnecessary details. The class **VoiceRecognition** provides this function. Two methods are the heart of this class:

1. **InitRecognize** – initializes voice recognition session

```
PROTECT METHOD InitRecognize (ptrHandle AS PTR, cVoiceName AS STRING,;
  dwConnectId AS DWORD) AS VOID STRICT CLASS VoiceRecognition
// PURPOSE:
//   Initialize voice recognition session
// PARAMETERS:
//   ptrHandle    –  handle to the window that controls the voice session
//   cVoiceName   –  name of the session
//   dwConnectId  -  ID number of the connection to the speech engine

LOCAL iSmc AS SHORTINT
LOCAL pReply AS PTR
LOCAL szVersion AS PSZ
LOCAL DIM aSmArg[10] IS SmArg

* check API version
SELF:iRc := SmApiVersionCheck (String2Psz (SM_API_VERSION_STRING),;
  @szVersion)
IF SELF:iRc = SM_RC_OK

  * Voice SmApiVersionCheck OK
  iSmc := 0
```

```
   * pass session and connection parameters to the speech engine
   SmSetArg (@aSmArg[++iSmc], String2Psz(SmNapplicationName),;
      String2Psz(cVoiceName))
   SmSetArg (@aSmArg[++iSmc], String2Psz(SmNwindowHandle), ptrHandle)
   SmSetArg (@aSmArg[++iSmc], String2Psz(SmNconnectionId),;
      PTR (_CAST, dwConnectId))

   * open SMAPI
   SELF:iRc := SmOpen (iSmc, @aSmArg)
   IF SELF:iRc = SM_RC_OK

      * Voice Open OK
      iSmc := 0

      * pass other arguments
      SmSetArg (@aSmArg[++iSmc], String2Psz(SmNrecognize),;
         PTR (_CAST, TRUE))
      SmSetArg (@aSmArg[++iSmc], String2Psz(SmNuserId),;
         String2Psz (SM_USE_CURRENT))
      SmSetArg (@aSmArg[++iSmc], String2Psz(SmNtask),;
         String2Psz (SM_USE_CURRENT))
      SmSetArg (@aSmArg[++iSmc], String2Psz(SmNenrollId),;
         String2Psz (SM_USE_CURRENT))

      * connect to the speech engine
      SELF:iRc := SmConnect (iSmc, @aSmArg, @pReply)
      IF SELF:iRc = SM_RC_OK

         * Voice Connect OK
         IF SELF:lDoPlay

            * request saving audio input
            SELF:iRc := SmSet (SM_SAVE_AUDIO, 1, @pReply)
            IF SELF:iRc = SM_RC_OK

               * Voice SmSet OK
            ELSE
               SELF:VoiceError ("SmSet")
            ENDIF
         ENDIF

         * enable dictation
         SELF:EnableDictation()
      ELSE
         SELF:VoiceError ("SmConnect")
      ENDIF
   ELSE
      SELF:VoiceError ("SmOpen")
   ENDIF
ELSE
   SELF:VoiceError ("SmApiVersionCheck")
ENDIF
```

2. **Recognize** – recognize the next voice input; i.e., a word, a phrase or dictation text

```
METHOD Recognize (lParam AS LONG) AS VOID STRICT CLASS VoiceRecognition
// PURPOSE:
//   Recognize next voice input
// PARAMETERS:
//   lParam  –  lParam ACCESS of the Event object passed as a parameter to the
//              Dispatch method of the controlling window

LOCAL dwAnnWords, dwW, dwWrd AS DWORD
LOCAL iR AS SHORT
LOCAL sm_msg_type AS SHORT
LOCAL cWord AS STRING
LOCAL lAnnotationsOK AS LOGIC
```

```
LOCAL sm_msg AS PTR
LOCAL pWord AS SM_WORD PTR
LOCAL strWord AS SM_WORD
LOCAL pAnnotations AS SM_ANNOTATION PTR
LOCAL strAnnotations AS SM_ANNOTATION
LOCAL oAnnotation AS Annotation

* init
SELF:INewWord := FALSE
SELF:INewPhrase := FALSE
SELF:INewText := FALSE

* receive a message from a speech engine
SmReceiveMsg (lParam, @sm_msg)
SmGetMsgType (sm_msg, @sm_msg_type)
DO CASE
  CASE sm_msg_type = SM_RECOGNIZED_WORD

     * a word was recognized - get recognition flag
     SmGetRc (sm_msg, @iR)
     SELF:iRc := iR
     IF SELF:iRc = SM_RC_OK

        * recognized
        SELF:iRc := SmGetFirmWords (sm_msg, @dwWrd, @pWord)
        SELF:dwWords := dwWrd
        IF SELF:iRc = SM_RC_OK

           * number of words
           IF SELF:dwWords > 0

              * got a word - get structure
              strWord := pWord

              * recognize next word
              IF SELF:RecognizeNextWord()

                 * flags
                 SELF:INewWord := TRUE
                 IF SELF:lWasWord
                    SELF:dwPreviousInput := kPREVIOUS_Word
                 ENDIF
                 SELF:lWasWord := TRUE

                 * remember word
                 IF PszLen (strWord.spelling) = 0

                    * word not understood
                    SELF:cNextWord := NULL_STRING
                    SELF:sNextWordSym := NULL_SYMBOL
                    SELF:cVocabName := NULL_STRING
                 ELSE

                    * word is OK
                    SELF:cNextWord := Psz2String (strWord.spelling)

                    * symbol
                    SELF:sNextWordSym := String2Symbol (SELF:cNextWord)

                    * remember tag and word
                    SELF:AddWord (strWord.tag, SELF:cNextWord)

                    * get ready to start remembering text
                    SELF:ITextStart := TRUE

                    * vocab
                    SELF:cVocabName := strWord.vocab
                 ENDIF
              ENDIF
```

```
            ENDIF

            * alternate words
            ASize (SELF:aAlternates, 0)
          ELSE
            SELF:VoiceError ("SmGetFirmWords")
          ENDIF
        ELSE

          * Voice SM_RECOGNIZED_WORD Error
        ENDIF
      CASE sm_msg_type = SM_RECOGNIZED_PHRASE

        * recognized a phrase
        ..................
      CASE sm_msg_type = SM_RECOGNIZED_TEXT .AND. SELF:lDictationOn

        * dictation text
        ..................
      CASE sm_msg_type = ...

        * other options
        ..................
    ENDCASE
```

Vocabularies and Grammars

The speech engine can run in one of two modes:

- command and control – when it recognizes single words and/or phrases; this mode is used for data entry and for controlling the application
- dictation – when the user enters plain text; e.g., a letter or a memo.

Command and control mode is used to enter well-defined data, so vocabularies of possible words and grammars of possible phrases can be prepared in advance. These vocabularies and grammars improve recognition, as the speech engine has to choose from a limited number of words only, and allow limiting the recognized input to valid entries only. Vocabularies are simple lists of words that can be spoken; e.g., “welcome”, “what’s hot”, “close”, etc. Grammars are structured collections of words and phrases together with rules of interaction between them, written in SRCL (Speech Recognition Control Language) format. Grammars serve controlled entry of entire phrases, so the user can speak almost freely and the application can still interpret the input correctly. Detailed description of grammars and their use in IBM ViaVoice can be found in the *SMAPI Developer’s Guide* (2). An example of one of the grammars used in the sample program is CHARLIE.BNF; it allows asking questions in such a way that the application can easily interpret them and give proper answers:

```
<convers> = <greeting>? <question> is your <favour>? <nouns> .
<greeting> = hello | good morning | excuse me .
<question> = what:"_what"
        | which:"_which" .
<favour> = favourite:"_favour" .
<nouns> = name:"_name"
        | job:"_job"
        | conference:"_conf"
        | food:"_food"
        | dish:"_food"
        | company:"_comp" .
```

where words in angle brackets denote rules, | stands for OR, and words after a colon are so-called annotations (numbers or keywords) that simplify interpretation of the entered phrase.

Connecting to Text-to-Speech ActiveX

The ability to translate spoken words to character strings is only one side of voice communication. After receiving the voice input, the program must be able to confirm what has been received and to answer possible questions. These tasks are performed by the text-to-speech engine, which converts character strings to spoken words. IBM provides with its ViaVoice the ActiveX control called Virtual Voices, which not only speaks but also displays a face (there are several to choose from) that personalizes the voice.

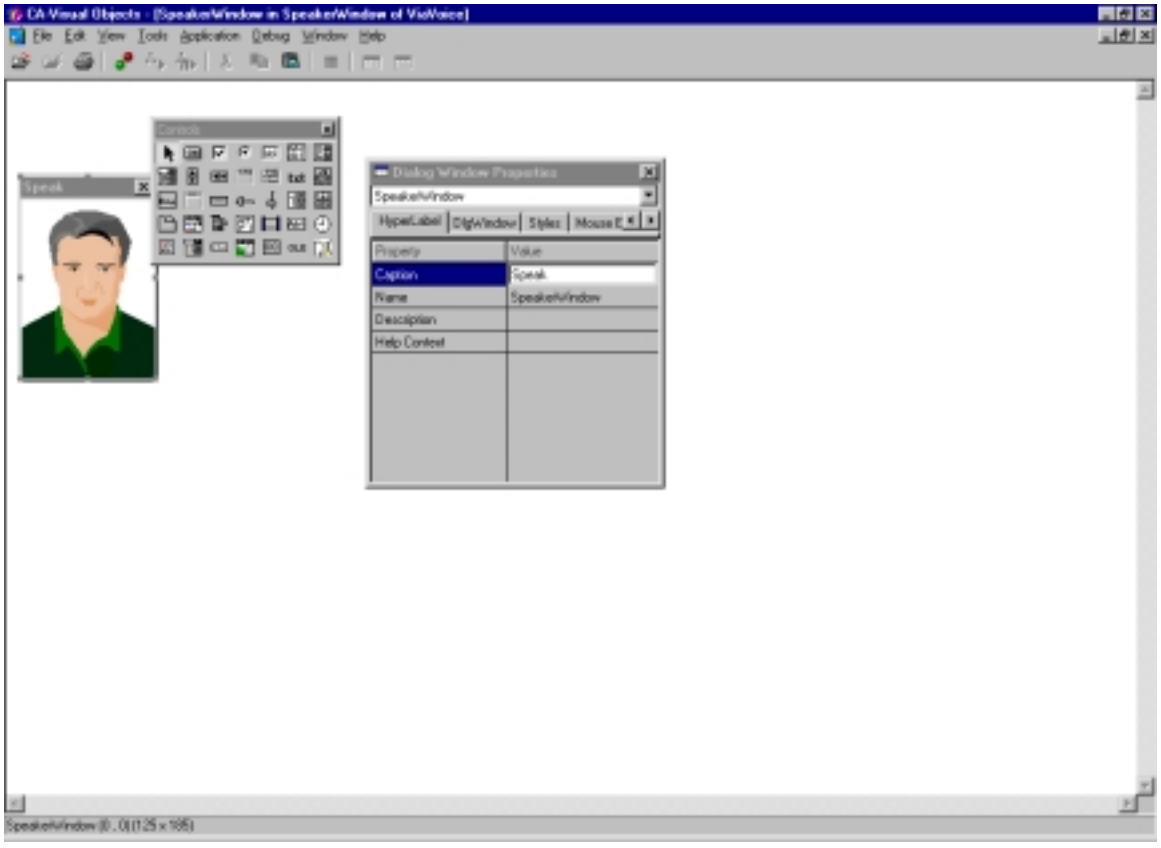
Connection to Virtual Voices is typical for accessing any ActiveX control from CA-Visual Objects. First the class **VirtualVoices**, which inherits from **OleControl**, is created. Then, the control inheriting from **VirtualVoices** is dropped on a window. In order to produce a voice output, it is enough to assign a character string to the **ASSIGN SpeakText** and then to call the **METHOD Speak()**.

The process of speaking can be controlled by two call-back methods:

- **StartSpeaking** – called just before the **VirtualVoices** control starts speaking
- **StopSpeaking** – called immediately after the **VirtualVoices** control stops speaking; this method is used to switch on the microphone after the message is said.

For the user's convenience, **SpeakerWindow,** a dialog window with a **VirtualVoices** control, can prepared. All communication to and from the text-to-speech control will be done via **SpeakerWindow**.

Fig 1. Designing the **SpeakerWindow** dialog window with CA-Visual Objects 2.5.



This window has to be enhanced with the following methods:

```
METHOD Close CLASS SpeakerWindow
// PURPOSE:
// Destroy the control when the window is closed
   IF SELF:oDCSpeaker != NULL_OBJECT

      * clean memory
      SELF:oDCSpeaker:Destroy()
   ENDIF

ACCESS Speaker CLASS SpeakerWindow
// pass the instance of the Virtual Voices control
   RETURN SELF:oDCSpeaker


METHOD SpeakerStopSpeaking( ) CLASS SpeakerWindow
// pass end-of speaking event to the parent window
   SELF:Owner:SpeakerStopSpeaking()
```

Simple Voice-Driven CA Store

The newest version of CA-Visual Objects has lots of interesting features. On the one hand, it can communicate with C-written DLLs like SMAPI.DLL and with ActiveX controls like Virtual Voices; on the other hand, it can access information stored in Jasmine ODB, Computer Associates object-oriented multi-media database. A well known example of Jasmine's capabilities is CA Store, which allows the user to walk through a set of fashion designs and select items to buy. So, let's try to enhance the CA Store example with voice recognition.

First, we have to add some extra functionality to **cDialoWindow**, the super window of all windows used by CA Store:

```
CLASS CDialoWindow INHERIT DialogWindow
   EXPORT aText AS ARRAY
   EXPORT aBit AS USUAL
   EXPORT oJClass AS JClass
   EXPORT oJCollection AS JCollection
   EXPORT oCurrentItem AS JObject

// new variables
   PROTECT lSpeaking AS LOGIC                  // speaker is speaking
   PROTECT oVoice AS VoiceRecognition          // voice recognition class
   PROTECT oParentWindow AS CdialoWindow       // parent window
   PROTECT oSpeakerWindow AS SpeakerWindow     // a window with a speaker
   PROTECT oSpeaker AS VIRTUALVOICES           // text-to-speech control

   EXPORT aVocabs AS ARRAY                     // vocabularies used by the
                                               // window
```

```
METHOD DisableVocabs CLASS CDialoWindow
// PURPOSE:
// Disable all vocabularies associated with the window

   LOCAL dwA, dwLen AS DWORD

   IF (dwLen := ALen (SELF:aVocabs)) > 0 .AND. SELF:oVoice != NULL_OBJECT
      FOR dwA := 1 UPTO dwLen
         SELF:oVoice:DisableVocab (SELF:aVocabs[dwA])
      NEXT
   ENDIF

METHOD EnableVocabs CLASS CDialoWindow
// PURPOSE:
// enable all vocabularies associated with the window

   LOCAL dwA, dwLen AS DWORD

   IF (dwLen := ALen (SELF:aVocabs)) > 0 .AND. SELF:oVoice != NULL_OBJECT
      FOR dwA := 1 UPTO dwLen
         SELF:oVoice:EnableVocab (SELF:aVocabs[dwA])
      NEXT
   ENDIF

METHOD SayAnswer (cAnswer) CLASS CDialoWindow
// PURPOSE:
// Send a character string to the Virtual Voices control
// PARAMETERS:
// cAnswer –  the message to be spoken

   * say message
   IF Empty (cAnswer)
      cAnswer := "I have nothing to say"
   ENDIF

// assign the text to the Virtual Voices control
   SELF:oSpeaker:SpeakText := cAnswer

// switch off the mocrophone
   SELF:oVoice:MicOff()

// say the assiged text
   SELF:lSpeaking := TRUE
   SELF:oSpeaker:Speak()

METHOD SpeakerStopSpeaking( ) CLASS CDialoWindow
// PURPOSE:
// Event handling method for end-of-speaking

// clean up after speaking – switch on the microphone
   SELF:oVoice:MicOn()
   SELF:lSpeaking := FALSE
```

All of the windows used in the example will inherit the above functionality.

Main Window

The next step is to decide which window will be the main voice controlling window, a window for the entire voice communication to go through. The natural choice is the **WHome** window. It has to be enhanced by the following methods:

```
METHOD PostInit(oParent,uExtra) CLASS WHome
// It is the original PostInit method, only with a call to PostInitVoice added
   SELF:BuildText()
   SELF:ControlHide()
   SELF:Pointer := pointer{POINTERHOURGLASS}
   SELF:SetBackGround( 'Background','Home', point{0,0}, SELF:Size,;
      Color{colorblack}, Color{colorRed} )
   SELF:Pointer := pointer{POINTERARROW}

   // the only new line – call to PostInitVoice
   SELF:PostInitVoice (oParent,uExtra)
 RETURN NIL

METHOD PostInitVoice(oWindow,uExtra) CLASS WHome
// PURPOSE:
// initiate voice recognition, define all vocabularies used by the
// application and display a window with a speaker
// PARAMETERS:
// oWindow    -  parent window
// uExtra     -  extra parameter

// create an instance of VoiceRecognition class to communicate with the
// voice engine:
//    SELF     -  a window that will communicate with the voice engine
//    "Test"   -  name of the voice recognition session
//    133      -  number of the voice recognition session used for identification
//                if the voice input
   SELF:oVoice := VoiceRecognition {SELF, "Test", 133, TRUE, TRUE, FALSE}


// define vocabularies for the WHome window:
//    VocabHome     -  a set of words that will be used to initiate actions
//    GrammHome     -  a grammar for a conversation with the Virtual Voices
//                     speaker – file CHARLIE.FSG is a compiled version of
//                     CHARLIE.BNF described above
   SELF:oVoice:DefineVocab ("VocabHome", {"welcome", "what's hot", "beauty",;
      "women", "men", "accessories", "table top", "history", "close"})
   SELF:oVoice:DefineGrammar ("GrammHome", "E:\CAWorld\CHARLIE.FSG")

// define vocabularies for the WWhatshot window:
//    VocabWhatshot   -  a set of words that will be used to initiate actions
   SELF:oVoice:DefineVocab ("VocabWhatshot", {"description", "home", "forward",;
      "show", "show next", "show previous", "previous", "order", "close"})
   SELF:oVoice:DefineGrammar ("GrammWhatshot", "E:\CAWorld\CHARLIE.FSG")

// define vocabularies for the other windows
.................

// start the voice recognition
   SELF:oVoice:Start()

// vacabularies used by this window
   SELF:aVocabs := {"VocabHome", "GrammHome"}

// enable only these vocabularies
   SELF:EnableVocabs()

// open a window with a speaker's face
   SELF:oSpeakerWindow := SpeakerWindow {SELF}
   SELF:oSpeakerWindow:Show()
```

```
// remember the instance of the Virtual Voices control
   SELF:oSpeaker := SELF:oSpeakerWindow:Speaker
   SELF:oSpeaker:ModeGuid := "{BF5EAD50-9F65-11CF-8FC8-0020AF14F271}"

// initial message spoken by Virtual Voices control
   SELF:SayAnswer ("Hi, I'm " + SELF:oSpeaker:ActorName +;
      ". Please start speaking")

// register axit to clean memory
   RegisterAxit (SELF)

// use own Dispatch method for event handling
   SELF:override()

METHOD Dispatch (oEvent) CLASS WHome
// PURPOSE:
// handle all window's events
// PARAMETERS:
// oWindow    -  event object

   LOCAL lProcessVoice AS LOGIC
   LOCAL dwParam AS DWORD

   // check if the event's message is WM_COMMAND – voice recognition engine
   // returns events with this message
   IF oEvent:Message = WM_COMMAND

      // get number of the voice recognition session (this is the number passed
      // to the voice engine while opening the session)
      dwParam :=oEvent:wParam
      IF dwParam = 133

         // the session ID is 133 (the number used above) so it is "our" event
         IF lProcessVoice := SELF:oVoice != NULL_OBJECT

            // recognize voice
            SELF:oVoice:Recognize (oEvent:lParam)
            IF SELF:oChildWindow != NULL_OBJECT

               // the event should be handled by a child window – pass it there
               lProcessVoice := !SELF:oChildWindow:VoiceInput()
            ENDIF
         ENDIF
         IF lProcessVoice

            // event not processed yet, so process it here
            SELF:VoiceInput()
         ENDIF
      ENDIF
   ENDIF

   // process all other events in a standard way
   RETURN SUPER:Dispatch(oEvent)

METHOD VoiceInput CLASS WHome
// PURPOSE:
// process voice events for this window

   LOCAL lProcessed AS LOGIC
   LOCAL cWord AS STRING
```

```
   // initialize
   lProcessed := FALSE
   DO CASE
      CASE SELF:oVoice:IsNewWord

         // a new word (from a vocabulary enabled by this window) for processing
         cWord := SELF:oVoice:NextWord
         IF SELF:oVoice:NextWordSym = NULL_SYMBOL

            // word unrecognized
            SELF:Caption := "Word not recognized"
         ELSE

            // word recognized OK – display it on the caption and on the control
            SELF:Caption := "Word recognized OK: " + cWord
            SELF:oDCVoiceInput:TextValue := cWord

            DO CASE
               CASE cWord == "welcome"

                  // the recognized word is "welcome"
                  // mark that it has been processed and
                  // call the appropriate method
                  lProcessed := TRUE
                  SELF:FixedTextClick (oDCFTWelcome)

               CASE cWord == "what's hot"
                  // the recognized word is "what's hot"
                  // mark that it has been processed and
                  // call the appropriate method
                  lProcessed := TRUE
                  SELF:FixedTextClick (oDCFTWhatsHot)

               // other words
               ......................
            ENDCASE
         ENDIF
      CASE SELF:oVoice:IsNewPhrase

         // a phrase from an enabled grammar has been recognized
         // display a new caption and the phrase
         SELF:Caption := "Phrase recognized OK"
         SELF:oDCVoiceInput:TextValue := SELF:oVoice:GetLastPhraseString

         // answer the phrase
         lProcessed := SELF:Answer()
   ENDCASE
   RETURN lProcessed

METHOD Answer CLASS WHome
// PURPOSE:
// generate answer to the voice input of a recognized phrase
// RETURN:
// TRUE    -  a phrase has been processed
// FALSE-  a phrase has not been processed

   LOCAL lProcessed AS LOGIC
   LOCAL lFavour AS LOGIC
   LOCAL dwLen, dwType, dwW, dwWordNo  AS DWORD
   LOCAL uAnnot AS USUAL
   LOCAL aAnnot, aPhrase AS ARRAY

   // initialize variables
   lFavour := FALSE
   lProcessed := FALSE

   //get the phrase and its annotations
   aPhrase := SELF:oVoice:GetLastPhrase
   aAnnot := SELF:oVoice:NextAnnotation
```

```
IF (dwLen := ALen (aAnnot)) > 0

   * there are annotations
   FOR dwW := 1 UPTO dwLen

      // get next annotation
      uAnnot := aAnnot[dwW]:Annotation

      //annotation type (generally it can be SYMBOL or DWORD)
      dwType := aAnnot[dwW]:Type

      // number of the word within a recognized phrase associated with the
      // processed annotation
      dwWordNo := aAnnot[dwW]:WordNo

      * process the annotations
      DO CASE
         CASE dwType = SYMBOL
            DO CASE
               CASE uAnnot = #_name

                  // phrase is a question about a name
                  IF lFavour

                     // word “favourite” has been used, so the name must be
                     // favourite
                     IF SELF:oSpeaker:ActorName == "Betty"

                        // the spearker is “Betty” so her favourite name is
                        // Charlie
                        SELF:SayAnswer ("My favourite name is Charlie")
                     ELSE
                        // the speaker is not “Betty” so the favourite name is
                        // Betty
                        SELF:SayAnswer ("My favourite name is Betty")
                     ENDIF
                  ELSE

                     // word “favourite” has not been used, so it is the
                     // question about the is speaker’s own name
                     SELF:SayAnswer ("My name is " + SELF:oSpeaker:ActorName)
                  ENDIF
                  lProcessed := TRUE
               CASE uAnnot = #_job

                  // the question is about the job
                  lProcessed := TRUE
                  SELF:SayAnswer ("I'm an actor")
               CASE uAnnot = #_favour

                  // word “favourite” has been used – set the flag
                  lProcessed := TRUE
                  lFavour := TRUE
               // other annotations
               ..........................
            ENDCASE
      ENDCASE
   NEXT
ENDIF
RETURN lProcessed
```

```
METHOD Close CLASS WHome
// PURPOSE:
// clean-up at the window's closing

   IF SELF:oSpeakerWindow != NULL_OBJECT

      // close the speaker window
      SELF:oSpeakerWindow:EndDialog()
   ENDIF

   IF SELF:oVoice != NULL_OBJECT

      // close the voice recognition class
      SELF:oVoice:Close()
   ENDIF
```

Child Windows

Child windows use the already active connection to the voice engine opened by **WHome**. Therefore, their **PostInitVoice** method must at least establish the two-way communication with the parent window, disable the parent window's vocabularies and enable the child window's own vocabularies:

```
METHOD PostInitVoice(oWindow,uExtra) CLASS WWhatshot
   // pass itself to the parent window
   oWindow:ChildWindow (SELF)
   // remember the parent window
   SELF:oParentWindow := oWindow

   // remember VoiceRecognition object
   SELF:oVoice := oWindow:Voice

   // disable the parent's vocabs
   SELF:oParentWindow:DisableVocabs()

   // enable own vocabs
   SELF:aVocabs := {"VocabWhatshot", "GrammWhatshot"}
   SELF:EnableVocabs()

   // remember SpeakerWindow and Speaker objects
   SELF:oSpeakerWindow := oWindow:SpeakerWindow
   SELF:oSpeaker := SELF:oSpeakerWindow:Speaker

   // make the speaker say something - optional
   SELF:SayAnswer ("Hi, I'm " + SELF:oSpeaker:ActorName)

   // register Axit
   RegisterAxit (SELF)

   // initialize dwPos
   SELF:dwPos := 1

METHOD Close CLASS WWhatshot
   // disable own vocabs
   SELF:DisableVocabs()

   // enable parent's vocabs
   SELF:oParentWindow:EnableVocabs()
```

As child windows communicate with ViaVoice through the **Dispatch** method of the main window, **WHome**, all voice output is returned there. Therefore, it is up to this method to redirect the voice input to the appropriate child window by calling its **VoiceInput** method:

```
METHOD VoiceInput CLASS WWhatshot
// PURPOSE:
// process voice events for this window

   LOCAL lProcessed AS LOGIC
   LOCAL cWord AS STRING

   // initialize
   lProcessed := FALSE
   DO CASE
      CASE SELF:oVoice:IsNewWord

         // new word
         cWord := SELF:oVoice:NextWord
         IF SELF:oVoice:NextWordSym = NULL_SYMBOL

            // word not recognized
            SELF:Caption := "Word not recognized"
         ELSE

            // word recognized OK
            SELF:Caption := "Word recognized OK: " + cWord

            DO CASE
               CASE cWord == "description"

                  // word “description” – open WDescription window
                  SELF:PBdescp()
                  lProcessed := TRUE
               CASE cWord == "home"

                  // return to WHome
                  lProcessed := TRUE
                  SELF:PBHome()

               // other words
               ........................
            ENDCASE
         ENDIF
      CASE SELF:oVoice:IsNewPhrase

         // a phrase from an enabled grammar has been recognized
         // display a new caption and the phrase
         SELF:Caption := "Phrase recognized OK"

         // answer the phrase
         lProcessed := SELF:Answer()
   ENDCASE
   RETURN lProcessed
```

In this example the grammar used by child windows is also CHARLIE.FSG, so the child’s **Answer** method is identical with the **Answer** method of the parent window.

Subclassing Windows and Controls

Adding voice communication to an application comes at the price of dealing with the complexity of communication between IBM's ViaVoice and CA-Visual Objects. However, the situation significantly improves when observing that it has to be done only once. The solution well known to OOP developers is writing window and control subclasses that will take care of the most laborious and complex tasks. All voice-aware windows and controls inherit from these subclasses.

The Window Subclass

Once the **VoiceRecognition** class has been implemented, it is time to place calls to it from a window that will communicate with the speech engine. The window subclass has to perform the following tasks:

1. Initialize the **VoiceRecognition** class.
2. Define window vocabularies, i.e., vocabularies with window control commands (e.g., "Close" for closing the window or control names to skip between controls).
3. Define control vocabularies and grammars (i.e., possible words to enter data to a control):

```
METHOD PostInit(oWindow,uExtra) CLASS VoiceDialogBasicWindow
// PUIRPOSE:
// Initialize voice recognition for a parent window

   // initialize VoiceBasic class that handles all tasks common to various
   // voice-enabled windows
   //   SELF                    -  a window that will communicate with the
   //                              voice engine
   //   SELF:cVoiceConnectName  -  name of the voice recognition session
   //   SELF:dwVoiceConnectID   -  number of the voice recognition session
   //                              used for identification of the voice input
   //   SELF:cMainDir           -  location of data dictionaries
   SELF:oBasic := VoiceBasic {SELF, SELF:cVoiceConnectName,;
      SELF:dwVoiceConnectID, SELF:cMainDir}
   IF SELF:oBasic:Success

      // initialization OK - remember
      SELF:oVoice := SELF:oBasic:Voice
      SELF:lDictation := SELF:oBasic:IsDictation

      // initialize no-pronunciation memory
      SELF:oBasic:AddCommandsInit()

      // generate a unique name for a vocabulary of control names
      SELF:cControlNameVocab := Symbol2String (SELF:NameSym) + "_CNTRL"

      // define the vocab
      SELF:oBasic:DefineExtraVocab (SELF, SELF:cControlNameVocab)

      // Dispatch method will be used to handle events
      SELF:Override()
   ELSE

      // quit
      SELF:EndDialog()
   ENDIF
```

```
METHOD Init (oWindow,cVoiceConnect,dwVoiceConnectID,cPath) CLASS VoiceBasic
// PURPOSE:
// Initialize VoiceBasic class that handles all tasks common to various
// voice-enabled windows
// PARAMETERS:
//   oWindow            -  window that will communicate directly with the
//                         voice engine
//   cVoiceConnect      -  name of the voice recognition session
//   dwVoiceConnectID-  number of the voice recognition session used
//                         for identification of the voice input
//   cPath              -  location of data dictionaries

   LOCAL aStandardVocabs AS ARRAY

   // remember
   SELF:cMainDir := cPath
   SELF:oOwner := oWindow

   // initialize variables
   SELF:lVoiceInOut := TRUE
   SELF:aVocabNames := {}
   SELF:oControlVocabs := ControlVocabs{}

   // names of vacabularies standard to all windows that will stay enabled
   // even when focus will move to another window
   aStandardVocabs := {"CNT_COMMON", "CNT_COMBO", "CNT_SLEDIT",;
      "CNT_NUMBER", "WIN_COMMON"}

   // read objects (window and control names) that will be used by during
   // the currently established voice recognition session
   IF SELF:ReadObjects (cVoiceConnect)

      // objects OK - add standard command vocabs
      SELF:aVocabNames := AddArrays (SELF:aVocabNames, aStandardVocabs)

      // read commands – words that will trigger events for windows and
      //                 controls
      IF SELF:ReadCommands()

         // read file names of vocabularies and grammars stored in external
         // files
         IF SELF:ReadVocabs()

            // check if dictation needed
            SELF:lDictation := SELF:IsItDictation()

            // initialize voice recognition class
            SELF:oVoice := VoiceRecognition {oWindow, cVoiceConnect,;
               dwVoiceConnectID, TRUE, TRUE, SELF:lDictation}

            // define extra vocabularies used by the window
            SELF:DefineVocabs()

            // define standard vocabularies used by the window
            SELF:DefineStandardVocabs (aStandardVocabs)
         ENDIF
      ENDIF
   ENDIF
```

For proper handling of controls, they have to be added to the window's memory by:

```
METHOD AddControl (sName AS SYMBOL, oControl AS Control,;
   cVoiceName := "" AS STRING) AS VOID STRICT CLASS VoiceDialogBasicWindow

   // add control to memory: name, control object and control's voice name
   // used to skip to it when spoke by the user
   SELF:oControls:Add (sName, oControl, cVoiceName)

   IF !SELF:AddControlNames (cVoiceName, Symbol2String (sName))

      // error in adding control names
   ENDIF

PROTECT METHOD AddControlNames (cWordList AS STRING, cCommand AS STRING);
   AS LOGIC STRICT CLASS VoiceDialogBasicWindow

   // add control's voice name to the list of window's commands
   // in the vocabulary which name is stored in SELF:cControlNameVocab
   RETURN SELF:oBasic:AddCommands (SELF:cControlNameVocab, cWordList,;
      cCommand)
```

As a child window does not have to handle communication with the voice engine directly, initialization is simpler:

```
   METHOD PreInit (oWindow, uExtra) CLASS VoiceDialogSubWindow

      // initialize
      SUPER:PreInit()

      // voice sub-window - remember voice recognition and basic handling
      // objects
      SELF:oVoice := oWindow:Voice
      SELF:oBasic := oWindow:Basic

   METHOD PostInit (oWindow, uExtra) CLASS VoiceDialogSubWindow

      LOCAL dwSubNo AS DWORD

      // register child window with the parent window so the voice input will
      // be redirected here
      dwSubNo := oWindow:OtherWindowRegister (SELF)

      // define vocab of control names
      SELF:cControlNameVocab := Symbol2String (SELF:NameSym) + "_CNTRL"_ +;
         NTrim (dwSubNo)


      // define extra vocabulary for the child window
      SELF:oBasic:DefineExtraVocab (SELF, SELF:cControlNameVocab)
```

4. Enable the defined window vocabularies while the owning window receives focus.

5. Keep enabling and disabling control vocabularies as the focus moves between controls; from the class **VoiceBasic**'s method **EditFocusChange** called from the window's method **EditFocusChange**:

```
METHOD EditFocusChange(oWindow, oControl, lGotFocus) CLASS VoiceBasic
// PURPOSE:
// Initialize VoiceBasic class that handles all tasks common to various
// voice-enabled windows
// PARAMETERS:
//   oWindow           -  window that will communicate directly with the
//                        voice engine
//   cVoiceConnect     -  name of the voice recognition session
//   dwVoiceConnectID-  number of the voice recognition session used
//                        for identification of the voice input
//   cPath             -  location of data dictionaries
```

```
   IF lGotFocus

      // control got focus
      IF IsMethod (oControl, #EnableVocabs)

         // method EnableVocabs exists
         IF oWindow:oControls = NULL_OBJECT

            // cannot track current control
            oWindow:dwCurrentControl := 0
         ELSE

            // find current control's number and remember it
            oWindow:oControls:Find (oControl:NameSym)
            oWindow:dwCurrentControl := oWindow:oControls:Pos
         ENDIF

         // enable control vocabs
         oControl:EnableVocabs()

         // remember current control
         oWindow:oCurrentControl := oControl
      ENDIF
   ELSE

      // control lost focus
      IF IsMethod (oControl, #DisableVocabs)

         // method DisableVocabs exists – use it
         oControl:DisableVocabs()
      ENDIF
   ENDIF
   RETURN NIL
```

6. Install the **Dispatch** method to:

   - receive events from the speech engine that is associated with voice input
   - mediate between the window and its controls, deciding which one receives the input
   - interpret the input, translating it to numbers, dates, strings, etc.

```
METHOD Dispatch (oEvent) CLASS VoiceDataBasicWindow
// PURPOSE:
// Receive events from Windows and process the speech engine events
// PARAMETERS:
// oEvent  –  Event object

LOCAL dwParam AS DWORD
LOCAL lProcessVoice AS LOGIC

IF oEvent:Message = WM_COMMAND

   // event comes from the speech engine - remember wParam
   dwParam := oEvent:wParam
   IF dwParam = SELF:dwVoiceConnectID

      // check if oVoice is instantated – it should be an object of the
      // VoiceRecognition class

      IF lProcessVoice := SELF:oVoice != NULL_OBJECT

         // recognize voice input
         SELF:oVoice:Recognize (oEvent:lParam)
         IF SELF:lOtherWindow
```

```
            // call the child window's VoiceInput
            lProcessVoice := !SELF:oOtherWindow:VoiceInput()
         ENDIF
      ENDIF
      IF lProcessVoice

         // voice input not yet processed - process it here
         SELF:VoiceInput()
      ENDIF
   ENDIF
ENDIF
RETURN SUPER:Dispatch(oEvent)

METHOD VoiceInput() AS LOGIC STRICT CLASS VoiceDialogBasicWindow
   RETURN SELF:oBasic:VoiceInput (SELF)

METHOD VoiceInput (oWindow AS OBJECT) AS LOGIC STRICT CLASS VoiceBasic
// PURPOSE:
// Process the voice input by calling appropriate methods for different types
// of input
// PARAMETERS:
// oWindow –  the controlling window
// RETURN
// TRUE when the input has been processed

LOCAL cVocabName AS STRING
LOCAL sWord AS SYMBOL
LOCAL oContrVocab AS ControlVocabElem

IF SELF:oVoice != NULL_OBJECT

   // name of the vocabulary where the recognized word comes from
   cVocabName := SELF:oVoice:VocabName
   DO CASE
      CASE SELF:oVoice:IsNewWord

         // new word
         sWord := SELF:oVoice:NextWordSym
         IF sWord = NULL_SYMBOL

            // word not recognized
            // call event handler for wird-not-recognized of the window
            oWindow:NoWord (NULL_STRING)
         ELSE

            // word recognized OK
            IF (oContrVocab := SELF:oControlVocabs:Find (cVocabName)) !=;
               NULL_OBJECT

               // control specific vocab – pass the word to the control
               oContrVocab:Control:VoiceInput (SELF:oVoice:NextWord)
            ELSE

               // not a control word; must be a word from window's vocabularies
               // call window's event handler for commands after translating
               // the word to an associated command
               IF !oWindow:ExtraCommands (SELF:GetAction (oWindow, sWord))

                  // not a window's command specific word
                  // call window's evenet handler for other words
                  oWindow:NewWord (SELF:oVoice:NextWord)
               ENDIF
            ENDIF
         ENDIF
      CASE SELF:oVoice:IsNewPhrase
```

```
         // phrase recognized
         DO CASE
            CASE cVocabName == kVOCAB_Number

               // phrase from a grammar for entering numbers
               // call window's event handler for numbers
               oWindow:NewNumber()
            CASE cVocabName == kVOCAB_NumberString

               // phrase from a grammar for entering number strings
               // (strings of digits and separators like dot, comma or dash)
               // call window's event handler for number strings
               oWindow:NewNumberString()
            CASE cVocabName == kVOCAB_Date

               // phrase from a grammar for entering dates
               // call window's event handler for dates
               oWindow:NewDate()
            OTHERWISE

               // non-standard phrase
               // call window's event handler for phrases
               oWindow:NewPhrase()
         ENDCASE
      OTHERWISE

         // voice input not processed
         RETURN FALSE
   ENDCASE
ENDIF
RETURN TRUE
```

7. Optionally perform supporting tasks, such as adding new words to vocabularies, training incorrectly recognized words, etc.

8. Perform cleaning before the window is closed ("undefine" vocabularies and clean the memory):

```
METHOD Close() CLASS VoiceDialogBasicWindow

   IF SELF:oCurrentControl != NULL_OBJECT

      // disable current control's vocabs
      SELF:oCurrentControl:DisableVocabs()
   ENDIF

   // disable window's vocabs
   SELF:DisableVocabs()

   IF SELF:oVoice != NULL_OBJECT

      // switch off the microphone
      SELF:oVoice:MicOff()
   ENDIF

   // close BasicVoice object
   SELF:oBasic:Close()
```

The Control Subclass

The tasks of a voice-enabled control can be summarized as follows:

1. Initialize control through a call to SUPER.
2. Initialize the **VoiceControl** class performing all voice-related tasks for the control.

   For example, a subclass of SingleLineEdit is initialized as follows:

```
METHOD Init (oOwner, uID, oPoint, oDimension) CLASS VoiceSingleLineEdit

   IF IsObject (uID)

      // from resource
      SUPER:Init (oOwner, uID)
   ELSE

      // dynamic
      SUPER:Init (oOwner, uID, oPoint, oDimension, EDITAUTOHSCROLL)
   ENDIF

   IF SELF:lUseVoice := oSettings_Global:lUseVoice

      // use voice
      // open VoiceControl class that handles all the voice-specific tasks
      // common to all controls
      SELF:oVoiceControl := VoiceControl {SELF}
   ENDIF
```

3. Enabling and disabling control vocabularies and grammars as the control gains or looses focus through calls to the appropriate methods of the **VoiceControl** class:

```
METHOD EnableVocabs() AS LOGIC STRICT CLASS VoiceControl
// PURPOSE:
// Enable vocabularies assigned to a control

   LOCAL dwVoc, dwW AS DWORD

   IF SELF:oVoice != NULL_OBJECT

      // enable all vocabs associated with the owning control
      IF (dwVoc := SELF:dwVocabLen) > 0
         FOR dwW := 1 UPTO dwVoc
            IF !SELF:oVoice:EnableVocab (SELF:aVocabName [dwW])
               RETURN FALSE
            ENDIF
         NEXT
      ENDIF
   ELSE
      RETURN FALSE
   ENDIF
   RETURN TRUE
```

4. Translating character strings to the control's commands.
5. Translating character strings to the desired input; e.g., substituting the recognized word for the control's value:

```
METHOD VoiceInput (cWord AS STRING) AS VOID STRICT CLASS VoiceSingleLineEdit
// PURPOSE:
// Enter the recognized word for the control's value
// PARAMETERS:
// cWord - the recognized word

   SELF:TextValue := cWord
```

Fully Voice-Driven CA Store

The simple voice-aware CA Store example presented earlier offered only voice control for window actions, such as opening or closing windows and displaying pictures. But with the complexity of dealing with controls, the programming would also become much more complex. The subclasses of windows and controls, however, solve the complexity problem, so now a complete voice-driven CA Store can be programmed easily.

Windows

Voice-aware windows inherit from the voice-aware window subclasses. Therefore, the inheritance originally common to all CA Store windows,

```
CLASS CDialoWindow INHERIT DialogWindow
```

must be replaced by two classes:

```
CLASS CMainDialoWindow INHERIT VoiceDialogBasicWindow
```

```
CLASS CNewDialoWindow INHERIT VoiceDialogSubWindow
```

These otherwise identical **CDialoWindow** classes will serve as superclasses to the CA Store voice-aware windows. The main window, **WHome**, inherits from **CMainDialoWindow** while other child windows inherit from **CNewDialoWindow**. Compared to its original version, the **WHome** window has now only a few changes:

```
METHOD PreInit (oParent,uExtra) CLASS WHome

   // set session parameters: connection ID number, name and windows name
   SELF:VoiceConnectID := 100
   SELF:VoiceConnectName := "CA_STORE"
   SELF:NameSym := #WIN_HOME

   // location of files
   SELF:cMainDir := WorkDir () + "\"

   SUPER:PreInit (oParent,uExtra)

METHOD PostInit(oParent,uExtra) CLASS WHome

   // as before
   ........................

   // the only new are these two lines
   SUPER:PostInit (oParent,uExtra)
   SELF:PostInitVoice (oParent,uExtra)
 RETURN NIL

METHOD PostInitVoice (oParent,uExtra) CLASS WHome

   // start recognition
   IF SELF:StartVoice()

      // open SpeakerWindow with text-to-speech control
      SELF:oSpeakerWindow := SpeakerWindow {SELF}
      SELF:oSpeakerWindow:Show()
      SELF:Speaker := SELF:SpeakerWindow:Speaker

      // inform other methods that speaker is in use
      SELF:UseSpeaker := TRUE
```

```
      // speaker's welcome message
      SELF:SayMessage ("Hi, I'm " + SELF:Speaker:ActorName +;
         ". Please start speaking")
   ELSE

      * error
      SELF:EndDialog()
   ENDIF

METHOD ExtraCommands (sCommand AS SYMBOL) AS LOGIC STRICT CLASS WHome
// PURPOSE:
// Event handler for performing non-standard commands specific to this window
// corresponeds to window's VoiceInput method from the previous example

   // display processed command in the window's caption
   SELF:Caption := "Command: " + Symbol2String (sCommand)
   DO CASE
      CASE sCommand == #WELCOME

         // call welcome
         SELF:FixedTextClick (oDCFTWelcome)
      CASE sCommand == #WHATS_HOT

         // open What's Hot window
         SELF:FixedTextClick (oDCFTWhatsHot)
      CASE sCommand == #BEAUTY
         ...................
      OTHERWISE

         // not an extra command – pass the command to SUPER
         RETURN SUPER:ExtraCommands (sCommand)
   ENDCASE
   RETURN TRUE

METHOD NewPhrase AS LOGIC STRICT CLASS WHome
// PURPOSE:
// Event handler for phrase recognition and associated actions
// corresponeds to window's Answer method from the previous example

   ...................
   RETURN SUPER:NewPhrase()

METHOD Close CLASS WHome
// last action before closing the window
   SELF:ThankYou()
   IF SELF:oSpeakerWindow != NULL_OBJECT

      * close speaker window
      SELF:oSpeakerWindow:EndDialog()
   ENDIF

METHOD ThankYou CLASS WHome
   SELF:SayMessage ("Thank you")
   SELF:WaitWhileSpeaking()
```

Notice that this window's programming is very similar to the one shown earlier. The real change can be seen when we consider a window that owns controls; there is not that much more programming and the effect is very different. For example, let's look at the **WOrder** window that owns SingleLineEdit controls:

```
CLASS WOrder INHERIT CNewDialoWindow

   // as before
   ................
```

```
METHOD PreInit (oParent,uExtra) CLASS WOrder

   // window's name
   SELF:NameSym := #WIN_ORDER

   // call SUPER
   SUPER:PreInit (oParent,uExtra)

METHOD PostInit(oParent,uExtra,uExtra1) CLASS WOrder

   // as before
   ..................

   // the only new are these two lines
   SUPER:PostInit (oParent,uExtra)
   SELF:PostInitVoice (oParent,uExtra)

METHOD PostInitVoice (oParent,uExtra) CLASS WOrder

   // add controls for subtotal, tax and total to the internal memory
   // all processing will be done automatically
   // for each control the following parameters must be passed:
   //    unique symbolic name
   //    instance variable of the control
   //    voice name – when spoken by the user, the focus will move to the
   //       associated control
   SELF:AddControl (#CNT_SUBTOT, oDCSleSubTotal, "subtotal")
   SELF:AddControl (#CNT_TAX, oDCSleTax, "tax")
   SELF:AddControl (#CNT_TOTAL, oDCSleTotal, "total")

   // start voice recognition
   IF SELF:StartVoice()

      // speaker is in use
      SELF:UseSpeaker := TRUE
   ELSE

      * error
      SELF:EndDialog()
   ENDIF

METHOD ExtraCommands (sCommand AS SYMBOL) AS LOGIC STRICT CLASS WOrder

   // handle window specific commands
   SELF:Caption := "Command: " + Symbol2String (sCommand)
   DO CASE
      CASE sCommand == #ACCEPT
         SELF:PBAccept()
      CASE sCommand == #CANCEL
         SELF:PBCancel()
      OTHERWISE

         // not an specific command – pass it to SUPER
         RETURN SUPER:ExtraCommands (sCommand)
   ENDCASE
   RETURN TRUE

METHOD NewNumber AS LOGIC STRICT CLASS WOrder
// PURPOSE
// Event handler for processing phrases for entry of numbers

   // process a number phrase
   // e.g. hundred twenty seven point four for entry of the value 127.4
   SUPER:NewNumber()

   // pass the recognized value to the current control
   SELF:oCurrentControl:Value := SELF:VoiceNumberCut
   RETURN TRUE
```

The method **NewNumber** will be called by the **VoiceInput** method of the **VoiceBasic** class when a phrase from the NUMBER grammar has been recognized. This grammar must be then enabled for all controls that require voice entry of a number.

The **WOrder** window looks as it did before, but this time the ViaVoice speaker appears on the screen and the user can control the application and enter data also by voice.

Fig. 2. Voice-aware WOrder window of CA Store.

More Developments

The developments described above open the way for the integration of voice communication within applications, thus enhancing their quality. However, the story would be incomplete without mentioning the need for further developments. As every person speaks a little bit differently than others from the same area and there are also quite big differences in the way people from various areas speak, ViaVoice performs best with typical speakers. For others, ViaVoice can be trained for improved recognition. Also, not all words are in ViaVoice's dictionaries yet. New words, e.g., names, can be added. Both problems can be addressed, but it is beyond the scope of this presentation.

Integration with Jasmine *ii*, brings benefits to both. Jasmine *ii* allows you to build the most natural computer applications, resembling well the world around us, full of pictures, voices, etc. ViaVoice allows for the most natural communication with the user. Working together, these two pieces of software bring computers even closer for all possible users.

Conclusion

Only recently, the advances in hardware and software made it possible to realize the old dream of communicating with computers the way we communicate with other people. However, the power comes at a price. A developer who wants to integrate speech communication into applications must master all of the internals of IBM's ViaVoice:

- initializing the speech engine
- defining and manipulating vocabularies and grammars
- translating character strings to the desired input
- handling of speech engine and Virtual Voices events.

CA-Visual Objects constitutes a development environment that significantly helps in the task, thanks to:

- the ability to access C-written DLLs and ActiveX controls
- fast database access and search functions that minimize the time of searching through vocabularies
- flexible event handling mechanisms
- powerful string processing functions
- OOP design that allows wrapping SMAPI functions and building window and control subclasses that can automatically handle typical voice communication tasks.

The benefits are high: the most natural way of communication, user friendliness, the ability to work without focusing attention on the screen and the keyboard, saving time in data entry, etc. These benefits also point to where such applications would be most useful: field applications, when the user has no possibility to access a computer, entering large amounts of data, free dictation, and online communication are only examples.

Bibliography

1. SMAPI (Speech Manager Application Programming Interface) Reference. IBM ViaVoice™ SDK for Windows®. Version 1.01

2. SMAPI Developer's Guide. IBM ViaVoice™ SDK for Windows®. Version 1.01

Marek Zawolek has been programming since the early seventies when he studied physics at The Jagiellonian University in Cracow, Poland. There, he gained experience with FORTRAN and PASCAL for numerical calculations, REDUCE for algebraic programming, and SNOBOL for character string processing. In 1989 he received his PhD degree from The Wageningen Agricultural University in The Netherlands, where he worked with computer simulation of plant diseases. From 1989 on, he was employed by Ciba-Geigy (today Novartis) in Egypt and in Switzerland, writing software for data collection and agricultural expert systems, first in CA-Clipper and later in CA-Visual Objects. Since January 1999 Marek is back in Poland, where he started Hexagram, a software company specialising in data entry and decision support systems written in CA-Visual Objects.

You can contact Marek at
e-mail: mzawolek@rasco.krakow.pl
CompuServe: 101743,2223