Prof. Bob is Ready to Answer Your Questions

Andrew Davison
`http://fivedots.coe.psu.ac.th/~ad/jg`
`ad@fivedots.coe.psu.ac.th`

25th July 2005, Draft #1

The ProfBob application is a mildly amusing example of speech synthesis, utilizing the freeTTS implementation of the Java Speech API (JSAPI).

Figure 1 shows the prodigiously talented Professor Bob ready for action:

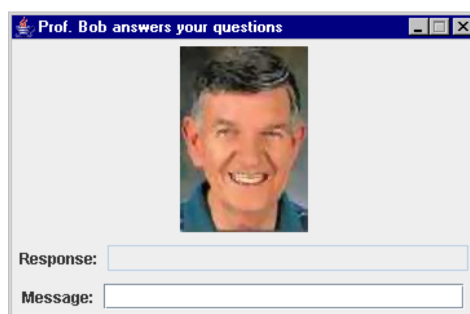

Figure 1. Prof. Bob is Ready.

The user enters a query into the "Message" text field, and Prof. Bob speaks a suitable response, while the text of that response appears in the "Response" textfield. Bob's mouth moves while the response is being spoken.

Aside from the use of speech synthesis, Prof. Bob is mostly "smoke and mirrors". His responses are generated by looking for predetermined keywords in the input text, and if nothing suitable is found, Bob falls back to vague rejoinders such as "Please go on". Bob's animated expression is achieved by moving a picture of his mouth up and down over his face.

In defense of this 'stage magic' style of programming, canned responses and simple graphics are arguably good enough for many kinds of user interaction (e.g. help desks, game avatars), and are easy to implement and extend. The alternative would be the grammatical parsing of the user's input, and 2D animation that emulates the large range of possible facial expressions.

The granddaddy of this type of computer-user interaction is *eliza*, a purely text-based system developed by Joseph Weizenbaum back in the 1960's, which handles input by simply rephrasing it. For example, "My head hurts" would generate the response "Why do you say your head hurts?", Details on eliza, including links to various implementations, can be found at the Wikipedia site, `http://en.wikipedia.org/wiki/ELIZA`.

© Andrew Davison 2005

Figure 2 shows the UML class diagrams for the ProfBob application, with the public methods visible:

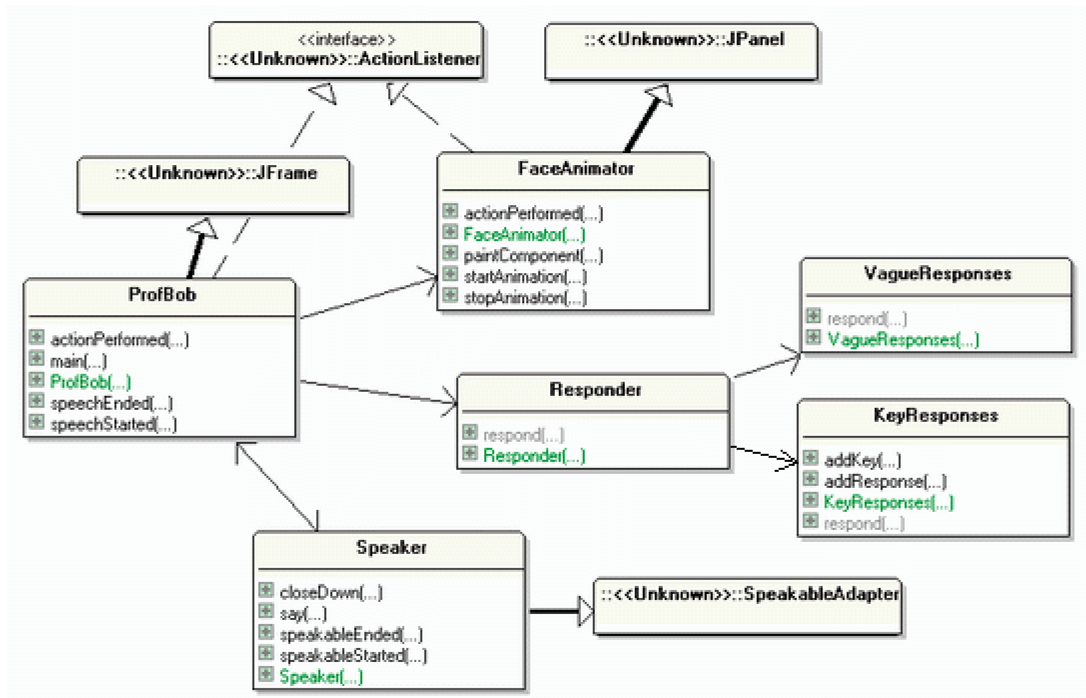

Figure 2. Class Diagrams for ProfBob.

The ProfBob class sets up the GUI, using FaceAnimator to animate the face, Responder to generate responses, and Speaker to synthesize speech. Responder utilizes KeyResponses objects to find keywords in the user's input and generate replies, and VagueResponses to serve up vague answers if no keywords are found. The data used by VagueResponses and the KeyResponses are read in from configuration text files at start-up.

The code for this application can be found in the ProfBob/ directory.

1. Dealing with the User

ProfBob starts by creating instances of the Responder and Speaker classes, and a FaceAnimator object (a subclass of JPanel) is added to the GUI.

ProfBob's main job is to mediate between the Responder, Speaker, and FaceAnimator objects; their interaction is triggered by the user typing a message into the jtfMsg textfield:

```
public void actionPerformed(ActionEvent e)
{
  if (e.getSource() == jtfMsg) {
    String msg = jtfMsg.getText().trim();    // get user's message
    if (!msg.equals("")) {
      jtfMsg.setEnabled(false);         // prevent any further input
      response = responder.respond(msg);    // get a textual response
      speaker.say(response);    // ask speaker to say the response
```


© Andrew Davison 2005

```
    }
  }
}  // end of actionPerformed()
```

The user's input is sent to the Responder object (responder) via a respond() call, and the result is passed to the Speaker object to be spoken.

Speaker's say() method returns immediately after adding the response string to a "speaking queue" associated with the synthesizer. Prof. Bob doesn't block waiting for the string to be uttered, which could potentially freeze the GUI for several seconds.

The Speaker object calls ProfBob's speechStarted() method when the synthesizer begins to say the response, and speechEnded() when the synthesizer finishes.

```
public void speechStarted()
{
  jtfResp.setText(response);   // show the response text
  faceAnim.startAnimation();   // animate the face
}


public void speechEnded()
{
  faceAnim.stopAnimation();    // stop the animation
  jtfMsg.setText("");          // clear the response field
  jtfMsg.setEnabled(true);     // prepare for new input
  jtfMsg.requestFocusInWindow();
}
```

When the speech begins, ProfBob instructs the FaceAnimator object (faceAnim) to start moving its mouth image up and down. When the synthesizer finishes the speech, faceAnim is told to stop the animation. This approach means that Bob's mouth movements are synchronized with the beginning and end of speech synthesis.

2. JSAPI and freeTTS

The Java Speech API (JSAPI, `http://java.sun.com/products/java-media/speech/`) covers two speech-related technologies: speech recognition, which converts spoken language into text, and speech synthesis that goes the other way, rendering text into spoken audio.

JSAPI is a specification, and Sun kindly invites interested parties to develop an implementation. Fortunately, several exist, including freeTTS (`http://freetts.sourceforge.net/`), which offers a fairly complete implementation of the speech synthesis components of JSAPI.

FreeTTS doesn't offer speech recognition, and the main missing synthesis feature is JSML processing (the Java Speech Markup Language). JSML can be employed to fine-tune pronunciation, stress, pauses, and other aspects of speaking, to make a computerized voice sound more natural.

FreeTTS comes with several built-in voices: I use the medium quality "kevin16" voice in ProfBob. Other voices can be added to the system.

 © Andrew Davison 2005

A list of JSAPI implementations can be found at `http://java.sun.com/products/java-media/speech/reference/codesamples/`. For speech recognition applications, I've used the freely available Sphinx-4 (`http://cmusphinx.sourceforge.net/sphinx4/`), which works pretty well, but you need a good microphone, and it helps to have an American accent :).

3. Bob Speaks

A speech synthesis application typically passes through 7 stages:

1) The desired properties for the synthesizer are collected together in a *mode descriptor*. The descriptor might include the name of the synthesizer that we require, its intended mode of use, and the chosen language.
2) The synthesizer engine is created, using the mode descriptor as a guide.
3) The synthesizer is moved to the ALLOCATED state, which causes it to load resources.
4) Once in the ALLOCATED state, various tweaks can be applied to the engine, such as changing its voice properties.
5) The synthesizer is moved into an ALLOCATION sub-state called RESUMED, which makes it ready to process text into speech. It's also a good idea to make sure that its speaking queue, where text is stored waiting for processing, is in the QUEUE_EMPTY state. This ensures that the supplied text will be processed as quickly as possible.
6) The text is passed to the engine, which renders it into speech
7) The synthesizer is closed down by moving it to the DEALLOCATED state, making it release its resources.

These stages are present in the Speaker class, but mostly hidden from external view. There are two public methods: say() which supplies text for processing, and closeDown() which terminates the synthesizer. say() implements stage 6, and closeDown() carries out stage 7.

The earlier stages, 1 to 5, are handled by initSynthesizer(), called from Speaker's constructor:

```
private void initSynthesizer()
{
  System.out.println("Synthesizer being initialized...");
  try {
    // specify the required synthesizer properties (stage 1)
    SynthesizerModeDesc desc =
        new SynthesizerModeDesc(
                null,          // engine name (don't care)
                "general",     // mode name -- general usage
                Locale.US,     // locale
                null,          // running (don't care)
                null);         // voice (none specified yet)
```

 © Andrew Davison 2005

```
    // create the synthesizer (stage 2)
    synthesizer = Central.createSynthesizer(desc);
    if (synthesizer == null) {
      System.err.println( noSynthMsg() );
      System.exit(1);
    }

    // allocate resources (stage 3)
    synthesizer.allocate();
    synthesizer.waitEngineState(Synthesizer.ALLOCATED);

    // modify syntheszier properties (stage 4)
    SynthesizerProperties synProps =
                 synthesizer.getSynthesizerProperties();
    synProps.setVoice( getVoice() );    // set the engine's voice
    synProps.setPitch(100);             // lower the pitch (deeper)
    // synProps.setSpeakingRate(100.0f);   // speak slower

    // get synthesizer ready to speak  (stage 5)
    synthesizer.resume();

    // wait until the synthesizer is ready to speak
    synthesizer.waitEngineState(Synthesizer.RESUMED);
    synthesizer.waitEngineState(Synthesizer.QUEUE_EMPTY);
    System.out.println("Synthesizer ready");
  }
  catch (Exception e)
  {  System.out.println(e);
     System.exit(1);
  }
}  // end of initSynthesizer()
```

The SynthesizerModeDesc class is used to create a mode descriptor. The null arguments in its constructor mean that the description doesn't care about the engine name, running behaviour, or voices that the synthesizer offers. The descriptor only requires that the synthesizer can manage general speech, in American English.

The several calls to Synthesizer.waitEngineState() force the Speech object to wait until the synthesizer enters a desired state. For example, the call to Synthesizer.resume () in stage 5 only *requests* that the synthesizer moves into the RESUMED state. The subsequent waitEngineState() calls waits for that state to be attained.

Once the synthesizer is in an ALLOCATED state, its behaviour can be adjusted using a SynthesizerProperties object. initSynthesizer() sets the voice, makes its pitch lower (deeper), and slows the speaking rate; the aim is to make the voice seem older and more authoritative. Actually, it often sounds more like drunkenness! That's why the Synthesizer.setSpeakingRate() call is commented out. The normal pitch for a male voice is about 130 Hz, with a speaking rate of about 200 words per minute. SynthesizerProperties settings are considered to be hints, so may have no effect, depending on the synthesizer.

3.1. Specifying Bob's Voice

The getVoice() method searches through the voices associated with the synthesizer, looking for the voice named VOICE_NAME ("kevin16"). If it's present then its

© Andrew Davison 2005

corresponding Voice object is returned, and subsequently employed by the synthesizer back in initSynthesizer().

```
// global constant
private static final String VOICE_NAME = "kevin16";


private Voice getVoice()
// get the Voice object associated with the VOICE_NAME name
{
  // get the properties for this synthesizer engine
  SynthesizerModeDesc desc =
          (SynthesizerModeDesc) synthesizer.getEngineModeDesc();
  Voice[] voices = desc.getVoices();

  // check if VOICE_NAME is a known Voice for this engine
  Voice voice = null;
  for(int i = 0; i < voices.length; i++) {
    if (voices[i].getName().equals(VOICE_NAME)) {
      voice = voices[i];
      break;
    }
  }
  if (voice == null) {
    System.err.println("Synthesizer could not find the " +
                         VOICE_NAME + " voice");
    System.exit(1);
  }

  // make the voice older
  voice.setAge(Voice.AGE_OLDER_ADULT);   // 60+ in age

  return voice;
} // end of getVoice()
```

"kevin16" is one of two general-purpose American voices offered by freeTTS (the other being the lower quality "kevin"). Other voices can be added to freeTTS's repertoire.

The voice is made older, by calling its setAge() method. Unfortunately, this has no noticeable effect upon "kevin16".

3.2. Speaking a Response

The say() method is called by ProfBob to request that a message is spoken.

```
public void say(String msg)
{
  try {
    synthesizer.speakPlainText(msg, this);  // add to speaking queue
  }
  catch (Exception e)
  {  System.out.println(e); }
}
```

 © Andrew Davison 2005

speakPlainText() returns immediately after adding the text to the synthesizer's speaking queue, which means that the GUI won't block while a potentially long sentence is spoken. This leaves the question of *when* to start and stop the face animation.

The solution is to have the Speaker object listen for JSAPI *events*, generated as the text is being rendered into speech.

The second argument of speakPlainText() is a reference to Speaker, which extends the SpeakableAdapter class (see Figure 2). SpeakableAdapter implements the SpeakableListener interface with empty methods for processing SpeakableEvent events. SpeakableEvent events can be utilized to monitor the progress of the spoken output.

Speaker must detect when speech begins and ends, so the face animation can be started and stopped. The most useful SpeakableEvent events for this are SPEAKABLE_STARTED and SPEAKABLE_ENDED, which are handled by the SpeakableListener methods speakableStarted() and speakableEnded(). Their implementations in Speaker are:

```
public void speakableStarted(SpeakableEvent e)
{  bob.speechStarted();  }

  public void speakableEnded(SpeakableEvent e)
{  bob.speechEnded();  }
```

The calls to speechStarted() and speechEnded() prod ProfBob into action at the required moments.

This mechanism works well almost all the time, with the animation and speech being closely synchronized. The only problem is when ProfBob speaks for the *first* time – in that case, the face animation starts almost a second before the voice. The fault seems to lie with the event mechanism in freeTTS: a SPEAKABLE_STARTED event is emitted well before any sound is generated, and this early event triggers an early animation.

3.3. Better Coordination of the Animation and Speech

A drawback with Speaker is that there's no coordination between the face animation and speech *while* the speech is being rendered: the words in the response have no effect on mouth movement.

This could be remedied by using another SpeakableEvent event, WORD_STARTED, which is output whenever a new word is encountered in the text. This event is handled by the SpeakableListener method wordStarted(). Unfortunately, freeTTS doesn't support this event.

Yet another approach is to extract the phonemes from the text, and use them to create a sequence of movements for the mouth image. Indeed, the phonemes could be employed to select different mouth images to more accurately reflect how a sound is pronounced. A useful method for this technique is Synthesizer.phoneme(), which converts text into a string of phonemes. Unfortunately, freeTTS doesn't offer an implementation for phoneme() either.

 © Andrew Davison 2005

3.4. Closing Down the Synthesizer

As ProfBob is closed, it calls Speaker's closeDown() method to terminate the synthesizer:

```
public void closeDown()
{
  try {
    synthesizer.cancel();         // cancel any speaking
    synthesizer.deallocate();     // free the engine's resources
  }
  catch (Exception e)
  {  System.out.println(e); }
}
```

Any currently executing speech is cancelled before the resources are deallocated; this corresponds to stage 7 in the list above.

4. Draw-dropping Action

FaceAnimator is a JPanel containing a face image of Prof. Bob, and a mouth picture which is drawn over the top. The images used in ProfBob are shown in Figure 3.

Figure 3. Prof. Bob's Face and Mouth.

FaceAnimator stores these images in two ImageIcons, faceIm and mouthIm. The size of the FaceAnimator panel is set equal to the size of the face.

I copied the mouth from the face image, making a note of its top-left (x,y) corner, which is used to position it on screen at run time. I also noted the image's height, which is a useful guide when deciding on the maximum vertical movement value for the mouth. These values are stored as constants in FaceAnimator:

```
// normal position for the mouth
private static final int MOUTH_X = 39;
private static final int MOUTH_Y = 101;

// the offset downwards will be between 1 and MAX_OFFSET+1
private static final int MAX_OFFSET = 2;
```

 © Andrew Davison 2005

4.1. Moving the Mouth

FaceAnimator starts moving the mouth up and down when its startAnimation() method is called, and stops the movement when stopAnimation() is executed. These methods are called by ProfBob to coincide with the start and end of speech synthesis.

The animation is controlled by a local Timer which is started (or resumed) by startAnimation() and paused by stopAnimation().

```
// globals
private static final int DELAY = 100;  // in ms, for the timer

private Timer animTimer = null;         // animation timer
private boolean stopAnimation = false;  // to pause the timer


public void startAnimation()
{
  stopAnimation = false;
  if (animTimer == null) {
    animTimer = new Timer(DELAY, this);   // start the timer
    animTimer.start();
  }
  else if (!animTimer.isRunning())
    animTimer.restart();    // resume the timer
} // end of startAnimation()


public void stopAnimation()
{  stopAnimation = true;  }
```

stopAnimation() doesn't directly stop the animation, only sets the stopAnimation boolean to true.

As the timer ticks away, actionPerformed() is called repeatedly. It updates the offset value for the mouth and requests a redraw, unless stopAnimation is true.

```
// globals for controlling the mouth movement
private boolean isOffset = false;
private int offset = 0;    // offset down from mouth's normal posn


public void actionPerformed(ActionEvent e)
{
  if (stopAnimation && !isOffset)   // mouth not offset, so can stop
    animTimer.stop();    // pause the timer
  else {
    offset = getMouthOffset();
    repaint();
  }
}
```

The stopAnimation boolean only makes the timer stop if isOffset is false. This boolean records whether the mouth image is currently offset downwards or not. The idea is that the mouth should only stop moving when it's back in its starting position (i.e. don't leave Bob with his mouth gaping open). This means that the call to

 © Andrew Davison 2005

stopAnimation() may take a short time to be acted upon, since the mouth must return to its initial position before the timer is stopped.

The mouth's offset is randomly generated by getMouthOffset(), with no relationship to what's being said. Surprisingly, this isn't particularly noticeable at execution time.

```
private int getMouthOffset()
{ int offset;
  if (!isOffset) {  // not offset, so can move the mouth down
    offset = rand.nextInt(MAX_OFFSET) + 1;
    isOffset = true;
  }
  else {   // mouth is offset, so draw it back at its original posn
    offset = 0;
    isOffset = false;
  }
  return offset;
}  // end of getMouthOffset()
```

getMouthOffset() calculates an offset downwards for the mouth, but alternates between an offset and no offset. This makes the mouth move down and up repeatedly, with the mouth returning to its starting position at regular intervals. The offset is a random integer between 1 and MAX_OFFSET+1.

4.2. Drawing the Face and Mouth

The call to repaint() at the end of actionPerformed() triggers a repaint of the JPanel, which involves drawing the face and the offset mouth on top of it.

```
public void paintComponent(Graphics g)
{
  super.paintComponent(g);
  faceIm.paintIcon(this, g, 0, 0);   // draw the face
  mouthIm.paintIcon(this, g, MOUTH_X, MOUTH_Y + offset);  // mouth
}
```

5. Deciding on a Response

Responder's respond() method is called by the ProfBob to generate a suitable textual response for the user's message.

Responder utilizes two approaches: first it passes the message to a series of KeyResponses objects (stored in an ArrayList). Each object takes turns looking for a keyword, and if one is found, the object generates a response.

If none of the KeyResponses objects find a keyword then Responder calls on a VagueResponses object, which serves up a response independent of the input message. Consequently, the response is rather 'vague'.

The ArrayList of KeyResponses objects is created by Responder reading in lines from a configuration text file, keyResponses.txt, which consists of groups of keywords and responses delimited by #K and #R markers.

For example, keyResponses.txt contains:

© Andrew Davison 2005

```
#K
java c++ c# j2me j2ee perl php math

#R
$ is a tool, much like a hammer.
Learning $ is just the first step.
Try typing $ into Google; it's what I do.

#K
exercise lab question exam midterm
program homework project

#R
Why do you hate the $?
I hope you're not working on the $ with other students?
```

Each paired group of keyword and response lines is used to create a new KeyResponses object. For instance, the example above would result in two KeyResponses objects: the first #K and #R blocks initialize the first object, the second pair of #K and #R lines instantiate the second object.

A response line may contain a '$', in which case the keyword that triggers a match is inserted in place of the '$' before the response is returned.

Although the KeyResponses class isn't 'intelligent' in any sense, it does utilize domain-specific knowledge in the form of its configuration file. The choice of keywords, and the separation of keywords and responses into distinct groups reflects the developer's knowledge. For instance, the first group of keywords in the example above all relate to programming languages, and their responses are suitable for that topic.

The VagueResponses object is responsible for its own initialization, reading in lines from a text file called vagueResponses.txt. It contains lines such as:

```
Let's get down to business.
Please elaborate on that a little more.
Please go on.
```

One of these lines is served up at semi-random when the Responder object calls VagueResponses' respond(). The response is semi-random in the sense that it will be randomly selected, but will not be the response returned by the previous call to respond(). This prevents Bob from repeating himself too frequently.

© Andrew Davison 2005