Level: Intermediate
Works with: Sametime
Updated: 27-Oct-2003

by
Gray
Norton

In Part 1 of this article series, we introduced you to interactive agents, a class of applications that includes Lotus Instant Messaging (Sametime) bots. We provided an overview of how you can create Lotus Instant Messaging interactive agents with the BuddyScript SDK (Software Developer's Kit) from ActiveBuddy. In Part 2, we get down to the details, stepping you through the process of developing an interactive agent that helps users manage a stock portfolio. (You can download the complete sample file from the Sandbox.)

This article assumes that you're familiar with basic instant messaging concepts. Some experience with Web software development is helpful.

A simple interactive agent for stocks

You've seen how simple it is to build an interactive agent with the BuddyScript SDK in the first part of this article series. Now let's delve into a real-world example. This article looks at the complete script for an interactive agent named Stocky that is designed to provide stock quotes and related functionality. In addition to fetching current quotes at the user's request, our interactive agent maintains a portfolio of favorite stocks for each user and watches for stocks to reach target values as specified by the user.

The following is a sample session transcript, logging a user's conversation with the Stocky interactive agent:

JohnDoe: hi
Stocky: Welcome, JohnDoe!

Type "help" to see what I can do!

JohnDoe: help
Stocky: I can:
1 Give you stock quotes
2 Manage a portfolio of your favorite stocks
3 Notify you when stocks hit target values

>> Type "help" at any time to return to this menu...

JohnDoe: give me a quote for ibm
Stocky:

```
Stock: IBM
Now: 81.80
Chg: -0.74
Hi/Lo: 82.88/81.60
Vol: 7.28 Million

>> Type "add" to add IBM to your portfolio
```

JohnDoe: add
Stocky: Added. Your portfolio contains:

```
Stock: IBM
Now: 81.80
Chg: -0.74
Hi/Lo: 82.88/81.60
Vol: 7.28 Million
```

JohnDoe: add ge, gm, and mmm
Stocky: Added. Your portfolio contains:

```
1 IBM: 81.80 -0.74
2 GE: 27.90 -0.47
3 GM: 36.91 -0.34
4 MMM: 139.55 -1.43
```

JohnDoe: tell me if ibm drops to 80 or hits 90
Stocky: OK, I will notify you when...
IBM falls to or below 80
IBM climbs to or above 90

JohnDoe: thank you!
Stocky: You're welcome!

Note that our sample application pulls its stock data from Yahoo! and is intended for learning and demonstration purposes only. *You should not deploy an interactive agent using this data in a production environment without first securing Yahoo!'s permission.*

Getting the data

Our script starts with code for interfacing with external data (in this case, Yahoo!'s Web-based stock quote service) and a pair of text files containing the stock symbols for all companies traded on the NYSE and NASDAQ exchanges:

```
package lib:/DataTypes/DateTime
package lib:/English/English

####################
# Live Stock Data #
####################

macro STOCK_COLUMNS
Symbol, Current, Date, Time, Change, Open, High, Low, Volume

macro STOCK_REGEX
\"(.*?)\",(.*?),\"(.*?)\",\"(.*?)\",(.*?),(.*?),(.*?),(.*?),(.*?)\s\n(.*)

// Builds a "+"-delimited string of stock symbols (required
// by the Yahoo! stock quote service) from a BuddyScript
```



```
// object variable.
function SymbolString(STOCK_LIST)
   STRING = ""
   or SYMBOL in STOCK_LIST
      STRING = StringConcat(STRING, SYMBOL, "+")
   return STRING

datasource GetStockInfo(SYMBOL_LIST) => MACRO_STOCK_COLUMNS {expire="now"}
   preprocess
      SYMBOL_STRING = SymbolString(SYMBOL_LIST)
   http
      http://finance.yahoo.com/d/quotes.csv?s=SYMBOL_STRING&f=sl1d1t1c1ohgv&e=.txt
   buddyscript
      R.Symbol, R.Current, R.Date, R.Time = {}, {}, {}, {}
      R.Change, R.Open, R.High, R.Low, R.Volume = {}, {}, {}, {}, {}
      MATCHES = {}
      while StringMatch(DATA, "MACRO_STOCK_REGEX", "s", MATCHES) > 0
         insert last in R.Symbol MATCHES[0]
         insert last in R.Current MATCHES[1]
         insert last in R.Date MATCHES[2]
         insert last in R.Time MATCHES[3]
         insert last in R.Change MATCHES[4]
         insert last in R.Open MATCHES[5]
         insert last in R.High MATCHES[6]
         insert last in R.Low MATCHES[7]
         insert last in R.Volume MATCHES[8]
         DATA = MATCHES[9]
      R.Offset = 0
      R.Count = Count(R.Symbol)
      return R

##################
# Stock Symbols #
##################

datatable NASDAQSymbols
   load Company {index=thawed}, Symbol {index=case-insensitive}, Exchange from file
      NSD.txt

datatable NYSESymbols
   load Company {index=thawed}, Symbol {index=case-insensitive}, Exchange from file
      NYE.txt

subpattern ANasdaqSymbol get Symbol, Symbol in NASDAQSymbols {minlength=1 maxlength=1
score=MACRO_MEDIUM_SCORE}
subpattern AnNyseSymbol get Symbol, Symbol in NYSESymbols {minlength=1 maxlength=1
score=MACRO_MEDIUM_SCORE}

subpattern AStock
+ STOCK=ANasdaqSymbol
+ STOCK=AnNyseSymbol
   return STOCK

subpattern AListOfStocks
+ STOCK=AStock
+ STOCK_LIST=AListOfStocks [(and|or)] STOCK=AStock
   STOCK_LIST[STOCK] = 0
```



```
    return STOCK_LIST
```

The code for getting stock quotes takes the form of a BuddyScript datasource, a type of function specialized for getting data and returning it in the tabular format required to take advantage of BuddyScript's advanced presentation features. Throughout the script, this datasource is called whenever a stock quote is required. The stock symbols, on the other hand, are stored in BuddyScript datatables. A datatable is a structure used to keep data resident in memory. Keeping the symbols in memory allows them to become part of the interactive agent's "vocabulary," which we accomplish by defining subpatterns based on the content of the datatables.

Delivering quotes

Next, let's look at the code that implements the interactive agent's core functionality, delivering stock quotes in response to user requests. This code follows a convention used throughout the project: Application logic is broken into BuddyScript procedures, which are called by the routines that follow them in the script:

```
####################################
# Basic Stock Quote Functionality #
####################################

// Displays a detailed quote for a single stock.
procedure DisplayStockDetails(STOCK)
   - <blank/>
      <tab align="rl">
      Stock:|STOCK.Symbol
      Now:|Round(STOCK.Current, ".2")
      Chg:|Round(STOCK.Change, "+.2")
      Hi/Lo:|Round(STOCK.High, ".2")/Round(STOCK.Low, ".2")
      Vol:|Round(STOCK.Volume/1000000, ".2") Million
      </tab>

// Displays a table with quotes for multiple stocks.
procedure DisplayStockInfo(SYMBOL_LIST)
   STOCK <= GetStockInfo(SYMBOL_LIST) show 5
      if SYS.Data.Count = 1
         // Only one stock -- display all details.
         call DisplayStockDetails(STOCK)
      else
         // Multiple stocks -- display a table.
         * <blank/>
            <tab>
         - STOCK.Symbol:|Round(STOCK.Current, ".2") Round(STOCK.Change, "+.2")
         {call=DisplayStockDetails(STOCK)}
         * </tab>
            <ifmore>>>> Type "more" for the rest of your quotes...</ifmore>

// Builds a nicely formatted string of stock symbols (for
// display) from a BuddyScript object variable.
function DisplaySymbolString(SYMBOL_LIST)
   NUM = Count(SYMBOL_LIST)
   STR, COUNTER = "", 1
   for SYMBOL in SYMBOL_LIST
      STR = StringConcat(STR, SYMBOL)
      if COUNTER < NUM - 1
         STR = StringConcat(STR, ", ")
      else if COUNTER = NUM - 1
         STR = StringConcat(STR, " and ")
      COUNTER = COUNTER + 1
   return STR
```



```
declare procedure AddToPortfolio

// Recognizes user requests for stock quotes.
+ [(=HowIs|=HowAre)] STOCK_LIST=AListOfStocks [doing]
+ [(=WhatIs|=WhatAre)] [the] [(price|prices)] [of] STOCK_LIST=AListOfStocks
+ [quote] STOCK_LIST=AListOfStocks
   call DisplayStockInfo(STOCK_LIST)
   - <blank/>
      >> Type "add" to add DisplaySymbolString(STOCK_LIST) to your portfolio
   + add
      call AddToPortfolio(STOCK_LIST)
```

The preceding script contains two procedures for displaying stock data. One displays all available data for a single stock, and the other displays a subset of the available data for several stocks at once. Both procedures utilize BuddyScript's table functionality (invoked via the markup tags) to produce clean, readable layouts. The DisplayStockInfo procedure also uses BuddyScript's enumeration (auto-numbering) feature, which displays a number next to each stock in the table. You can access a more detailed quote for any stock by typing the corresponding number. Enumeration can be added to any output line by including a set of curly braces ({}) at the end of the line; the code between the curly braces indicates what action should be taken when the user types the number.

The routine at the end of this snippet also deserves a closer look for a couple of reasons. First, the pattern definitions in this routine refer to subpatterns that are defined in the BuddyScript libraries. The BuddyScript libraries come with BuddyScript Server and contain useful code for a variety of purposes. In the first two lines of our script, we included a couple of library packages, English and DateTime. The subpatterns we use here (=WhatIs, =HowAre, and so on) are defined in the English package and are used to recognize both the contracted (for example, what's) and the full (for instance, how are) forms of common question phrasings. Later, we'll use functions from DateTime to format dates and times for display.

The same routine also utilizes BuddyScript's dialog feature to offer the user a context-based shortcut. After displaying the information for the requested stocks, the agent prompts you to type add to add the stocks to your portfolio. In any other context, the simple statement add would not give the interactive agent enough information to act on, but it's enough in this case because the list of stocks is already known. The code defining a dialog looks just like the code for defining a routine, consisting of one or more pattern definitions followed by an indented block of code. The only difference is that the dialog definition is itself indented within a block of BuddyScript code, indicating the point in the script's execution where the dialog should occur.

Keeping a user's portfolio

At this point, it's worth noting that the interactive agent's core functionality is essentially complete. In less than 100 lines of code (excluding comments), we've written an interactive agent that will live happily on users' contact lists, delivering instant stock quotes upon request. The remaining code, composed of a few hundred more lines, adds some compelling extra functionality and some interface niceties to improve the user experience.

The following snippet contains the code for storing, manipulating, and displaying the user's portfolio—just a list of favorite stocks, in this case:

```
############################
# Portfolio Functionality #
############################

stored variable MY_PORTFOLIO = {}

// Displays quotes for all stocks in the user's portfolio.
procedure ShowPortfolio
   if MY_PORTFOLIO = {}
      - Your portfolio is currently empty.
```



```
        exit
    - Your portfolio contains:
    call DisplayStockInfo(MY_PORTFOLIO)

  // Adds one or more stocks to the user's portfolio,
  // then displays the modified portfolio.
  procedure AddToPortfolio(STOCK_LIST)
    ADDED = 0
    for SYMBOL in STOCK_LIST
      if !Exist(MY_PORTFOLIO[SYMBOL])
        MY_PORTFOLIO[SYMBOL] = 0
        ADDED = 1
    if ADDED
      // At least one stock was added.
      - Added. \c
    else
      // All of these stocks were already in the portfolio.
      - Already in your portfolio. \c
    call ShowPortfolio

  // Removes one or more stocks from the user's portfolio,
  // then displays the modified portfolio.
  procedure RemoveFromPortfolio(STOCK_LIST)
    REMOVED = 0
    for SYMBOL in STOCK_LIST
      if Exist(MY_PORTFOLIO[SYMBOL])
        remove MY_PORTFOLIO[SYMBOL]
        REMOVED = 1
    if REMOVED
      // At least one stock was removed.
      - Removed. \c
    else
      // None of these stocks were in the portfolio.
      - Not in your portfolio. \c
    call ShowPortfolio

  // Clears the user's portfolio, removing all stocks.
  procedure ClearPortfolio
    MY_PORTFOLIO = {}
    - OK. Your portfolio has been cleared.

  // Subpatterns useful for recognizing portfolio requests.
  ////////////////////////////////////////////////////////
  subpattern Clear
  + (clear|reset|delete|cancel)

  subpattern Stocks
  + (stock|company|ticker|symbol)
  + (stocks|companies|tickers|symbols)

  subpattern Portfolio
  + [my] (portfolio|stocks)
  ////////////////////////////////////////////////////////

  // Recognizes requests to display the user's portfolio.
  + [(=What|=Which)] [=Stocks] [(=Is|=Are)] [in] =Portfolio
  + [(=WhatIs|=WhatAre)] [in] =Portfolio
```

```
+ =ShowMeNoun<=Portfolio>
   call ShowPortfolio

// Recognizes requests to clear the user's portfolio.
+ =Clear =Portfolio
   call ClearPortfolio

// Recognizes requests to add stocks to the user's portfolio.
+ add STOCK_LIST=AListOfStocks [[to] =Portfolio]
   call AddToPortfolio(STOCK_LIST)

// Recognizes requests to remove stocks from the user's portfolio.
+ remove STOCK_LIST=AListOfStocks [[from] =Portfolio]
   call RemoveFromPortfolio(STOCK_LIST)
```

The main point of interest in this code is the use of a BuddyScript stored variable to keep the user's portfolio in a way that persists across multiple sessions. One of the components of BuddyScript Server is the user profile, a repository for storing and retrieving user-specific data; the values of stored variables are kept here. User profile data is keyed to a user's screen name and automatically retrieved each time a session begins, making the persistence of data extremely easy and virtually transparent to the developer.

Tracking stocks

The next section of code accounts for the bulk of the script. This code implements the watchlist functionality, which takes full advantage of the fact that the interactive agent operates in a messaging environment. It proactively notifies the user whenever a stock reaches a specified target value and uses presence awareness to store a notification for later delivery if the user is not on-line when a target is reached.

The code for the functionality itself is relatively straightforward and is thoroughly commented. In a nutshell:

- If there are items in a user's watchlist, the interactive agent checks every five minutes to see if any of the target values have been reached.
- Each time it checks the watchlist, the interactive agent checks the user's presence. Notifications are sent immediately if the user is active, but stored for later delivery if the user is off-line or away.
- If all of the target values on the watchlist are reached, the list is emptied and the interactive agent stops making periodic checks (until the user adds new items to the list).

The code performs the periodic checks by using BuddyScript's notification feature. A notification is essentially just a timer: When the timer goes off, a specified procedure is called. In this case, the notification triggers the CheckPresence procedure, which requests the user's presence status. The status comes back asynchronously (via the ABBuddyStatus procedure) which records the presence information and calls CheckWatchlist. After performing its duties, CheckWatchlist uses a new notification to restart the process (provided there are still items in the watchlist).

The code begins as follows:

```
############################
# Watchlist Functionality #
############################

macro UPDATE_INTERVAL
in 5 minutes

stored variable MY_WATCHLIST = {}
stored variable WATCHLIST_NOTIF_ID = -1
stored variable THIS_NOTIF_ID = -1
variable PRESENT = 0
```

The following two functions are used by the CheckWatchlist and ShowWatchlist procedures to generate display

text, based on the type of event being watched for.

```
function Reaches(TYPE)
   if TYPE eq "<"
      return "falls to or below"
   else
      return "climbs to or above"

function Reached(TYPE)
   if TYPE eq "<*"
      return "fell below"
   else
      return "climbed above"
```

Next, we check each item in a user's watchlist to see whether or not it is triggered by the stock's current value. This happens in two circumstances: whenever items are added to the watchlist and periodically (but only if there are active items on the watchlist). In the second case, the user's presence status is determined before checking the watchlist. When the user is away, no outputs are made, but any notifications of items triggered are stored to be output later when the user returns:

```
procedure CheckWatchlist
   if WATCHLIST_NOTIF_ID ne -1
      // If we're being called by the periodic timer...
      if THIS_NOTIF_ID ne WATCHLIST_NOTIF_ID
         //...and the timer has been "reset," then don't execute.
         exit
   if MY_WATCHLIST = {}
      // If the watchlist has been cleared, don't execute.
      exit
   REMAINING_WATCHLIST = {}
   ADDED, TRIGGERED = "", ""
   STOCK <= GetStockInfo(MY_WATCHLIST)
      for VALUE in MY_WATCHLIST[STOCK.Symbol]
         for TYPE in MY_WATCHLIST[STOCK.Symbol][VALUE]
            ORIG_TYPE = TYPE
            if TYPE eq "<*" or TYPE eq ">*"
               // If this item has already been triggered but not yet output
               // because the user was away...
               DATE_REACHED = MY_WATCHLIST[STOCK.Symbol][VALUE][TYPE]
               if PRESENT
                  //...then prepare a notice to be output, if the user is now present...
                  MSG - STOCK.Symbol Reached(TYPE) $VALUE on DATE_REACHED. Its current value is
                  $STOCK.Current.
                  TRIGGERED = StringConcat(TRIGGERED, "\n   ", MSG)
               else
                  //...or put the item back in storage if the user is still away.
                  REMAINING_WATCHLIST[STOCK.Symbol][VALUE][TYPE] = DATE_REACHED
               continue
            if TYPE eq "?"
               // If the user has just added this item and didn't explicitly specify
               // a type ("<" or a ">"), infer the type from the current price
               // and the target value.
               if STOCK.Current = VALUE
                  // If the current price is equal to the target value, then the item
                  // is triggered.
                  CONDITION_MET = 1
               else
```



```
                // Otherwise, set the type.
                CONDITION_MET = 0
                if STOCK.Current > VALUE
                   TYPE = "<"
                else
                   TYPE = ">"
             else
                // If we already know the type, just check to see if the item is triggered.
                CONDITION_MET = if(TYPE eq ">", STOCK.Current >= VALUE, STOCK.Current <= VALUE)
             if CONDITION_MET
                // If the item is triggered...
                if PRESENT
                   // ...and the user is present, then prepare a notice to be output.
                   MSG - STOCK.Symbol is now valued at $STOCK.Current per share.
                   TRIGGERED = StringConcat(TRIGGERED, "\n   ", MSG)
                else
                   // If the user is away, store the item for later notification.
                   NOW = GetShortDisplayCurrentDate() "at" GetDisplayCurrentTime()
                   REMAINING_WATCHLIST[STOCK.Symbol][VALUE][StringConcat(TYPE, "*")] = NOW
             else
                // If the item is not triggered, put it back in the watchlist to be
                // checked again later.
                if MY_WATCHLIST[STOCK.Symbol][VALUE][ORIG_TYPE] = 0
                   // If this item has just been added, prepare a confirmation notice
                   // to be output.
                   MSG - STOCK.Symbol Reaches(TYPE) VALUE
                   ADDED = StringConcat(ADDED, "\n   ", MSG)
                REMAINING_WATCHLIST[STOCK.Symbol][VALUE][TYPE] = 1
   if ADDED
      // If any items were added, output the associated notices.
      - OK, I will notify you when...ADDED
   if TRIGGERED
      // If any items were triggered, output the associated notices...
      if ADDED
         // ...but break them out into separate messages if items were also
         // added during this pass.
         - <brk/>
      - Watchlist alert:TRIGGERED
   MY_WATCHLIST = REMAINING_WATCHLIST
   WATCHLIST_NOTIF_ID = notify MACRO_UPDATE_INTERVAL : CheckPresence
```

The following requests the user's presence status by adding the user to the interactive agent's contact list:

```
procedure CheckPresence
   THIS_NOTIF_ID = SYS.Notification.Id
   ABSendServiceEvent(addbuddy SYS.User.ScreenName)
```

These lines receive and store the user's presence status, remove the user from the contact list, and then check the user's watchlist:

```
procedure overrides ABBuddyStatus(EVENT)
   PRESENT = if(EVENT.status = "active", 1, 0)
   ABSendServiceEvent(removebuddy SYS.User.ScreenName)
   call CheckWatchlist
```

These three lines clear the user's watchlist, removing all items:


```
procedure ClearWatchlist
   MY_WATCHLIST = {}
   - Your watchlist has been cleared.
```

And these display the user's watchlist:

```
procedure ShowWatchlist
   if MY_WATCHLIST = {}
      - You are currently not watching any stocks.
      exit
   - I will notify you when...
   for SYMBOL in MY_WATCHLIST
      for VALUE in MY_WATCHLIST[SYMBOL]
         for TYPE in MY_WATCHLIST[SYMBOL][VALUE]
            WATCHLIST_OBJ = {}
            WATCHLIST_OBJ[SYMBOL][VALUE][TYPE] = 0
            - SYMBOL Reaches(TYPE) VALUE {call=RemoveFromWatchlist(WATCHLIST_OBJ)}
   - <blank/>
      >> To remove an item, type its number.
      >> To clear your entire watchlist, type "clear."
   + =Clear
      call ClearWatchlist
```

The following code adds one or more items to the user's watchlist:

```
procedure AddToWatchlist(WATCHLIST)
   for SYMBOL in WATCHLIST
      for VALUE in WATCHLIST[SYMBOL]
         for TYPE in WATCHLIST[SYMBOL][VALUE]
            MY_WATCHLIST[SYMBOL][VALUE][TYPE] = 0
   WATCHLIST_NOTIF_ID = -1
   PRESENT = 1
   call CheckWatchlist
```

And this removes one or more items from the user's watchlist:

```
procedure RemoveFromWatchlist(WATCHLIST)
   REMOVED = 0
   for SYMBOL in WATCHLIST
   for VALUE in WATCHLIST[SYMBOL]
         for TYPE in WATCHLIST[SYMBOL][VALUE]
            if TYPE eq "?"
               for EVERY_TYPE in MY_WATCHLIST[SYMBOL][VALUE]
                  remove MY_WATCHLIST[SYMBOL][VALUE][EVERY_TYPE]
               REMOVED = 1
               continue
            if Exist(MY_WATCHLIST[SYMBOL][VALUE][TYPE])
               remove MY_WATCHLIST[SYMBOL][VALUE][TYPE]
               REMOVED = 1
         if MY_WATCHLIST[SYMBOL][VALUE] = {}
            remove MY_WATCHLIST[SYMBOL][VALUE]
      if MY_WATCHLIST[SYMBOL] = {}
         remove MY_WATCHLIST[SYMBOL]
   if REMOVED
      - Removed. \c
   call ShowWatchlist
```


The following are subpatterns useful for recognizing watchlist requests:

```
subpattern TellMe
+ (tell|notify|alert|im|contact|ping) me
+ let me know

subpattern Watchlist
+ [my] (watchlist|watch list)

subpattern ToExceed
+ [to] exceed
+ exceeds
+ [has] exceeded

subpattern ToRise
+ [to] (rise|climb)
+ (rises|climbs)
+ [has] (risen|climbed)

subpattern ToFall
+ [to] (fall|drop|dip)
+ (falls|drops|dips)
+ [has] (fallen|dropped|dipped)

subpattern ToBe
+ is
+ [to] (be|become)
+ [has] become

subpattern ToReach
+ [to] (reach|hit)
+ (reaches|hits)
+ [has] (reached|hit)

subpattern ToGet
+ [to] (go|get) to
+ (goes|gets) to
+ [has] (gone|gotten) to
```

These are specialized subpatterns for parsing a conversational watchlist request and outputting the BuddyScript object format utilized by the various watchlist-related procedures:

```
subpattern Reaches
+ =ToExceed
+ [=ToBe] (more|greater|higher) than
+ [(=ToGet|=ToBe)] above
+ =ToRise (above|to)
   return ">"
+ [=ToBe] (less|lower) than
+ [(=ToGet|=ToBe)] below
+ =ToFall (below|to)
   return "<"
+ =ToReach
+ =ToGet to
+ =ToBe [at]
   return "?"
```



```
// "10 20 30", "10, 20 or 30", etc...
subpattern AListOfFloats
+ FLOAT=AFloat
+ LIST=AListOfFloats [(and|or)] FLOAT=AFloat
   LIST[FLOAT] = 0
   return LIST

// "falls below 10", "climbs to 20 or 30", etc...
subpattern ATargetValue
+ [TYPE=Reaches] VALUES=AListOfFloats
   TYPE |= "?"
   for VALUE in VALUES
      TARGET[VALUE][TYPE] = 0
   return TARGET

// "falls below 10 or climbs to 20 or 30", etc...
subpattern AListOfTargetValues
+ TARGET=ATargetValue
   return TARGET
+ LIST=AListOfTargetValues [(and|or)] TARGET=ATargetValue
   for VALUE in TARGET
      for TYPE in TARGET[VALUE]
         LIST[VALUE][TYPE] = 0
   return LIST

// "MSFT CSCO 10 40", "if MSFT or CSCO drops to 10 or rises to 40", etc...
subpattern AWatchlistFragment {matching="canskip"}
+ [(if|=When)] STOCK_LIST=AListOfStocks TARGET_LIST=AListOfTargetValues
   FRAG = {}
   for SYMBOL in STOCK_LIST
      for VALUE in TARGET_LIST
         for TYPE in TARGET_LIST[VALUE]
            FRAG[SYMBOL][VALUE][TYPE] = 0
   return FRAG

// "MSFT 20 30 IBM 80 90", "if MSFT hits 20 or 30 or if IBM drops to 80 or exceeds 90", etc...
subpattern AWatchlist
+ FRAG=AWatchlistFragment
   return FRAG
+ LIST=AWatchlist [(and|or)] FRAG=AWatchlistFragment
   for SYMBOL in FRAG
      for VALUE in FRAG[SYMBOL]
         for TYPE in FRAG[SYMBOL][VALUE]
            LIST[SYMBOL][VALUE][TYPE] = 0
   return LIST
```

Finally, these lines process watchword requests:

```
// Recognizes requests to add items to the user's watchlist.
+ [=TellMe] WATCHLIST=AWatchlist
+ [watch [for]] WATCHLIST=AWatchlist
+ [add] WATCHLIST=AWatchlist
   call AddToWatchlist(WATCHLIST)

// Recognizes requests to display the user's watchlist.
+ [(=What|=Which)] [(stock|stocks)] [(=Is|=Are)] [on] =Watchlist
+ [(=WhatIs|=WhatAre)] [on] =Watchlist
```



```
+ [(=What|=Which)] [(stock|stocks)] [[=Am] i] watching
+ =ShowMeNoun<=Watchlist>
   call ShowWatchlist

// Recognizes requests to clear the user's watchlist.
+ =Clear =Watchlist
   call ClearWatchlist

// Recognizes requests to remove items from the user's watchlist.
+ =DoNot [=TellMe] WATCHLIST=AWatchlist
+ =DoNot [watch [for]] WATCHLIST=AWatchlist
+ (remove|=Clear) WATCHLIST=AWatchlist
   call RemoveFromWatchlist(WATCHLIST)
```

Given the simplicity of this process, you may well be wondering why there is so much code in this section, especially compared to the previous snippets. The answer is that we're striving for some bonus points in the watchlist functionality to illustrate how far a little extra effort goes toward improving an interactive agent's conversational abilities. Specifically:

- We have chosen to be sensitive to linguistic cues as to the type of target you want to set—for example, that you want to be notified when IBM *climbs* to 90 as opposed to when it *falls* to 90. In the typical case (where you know what the current value of the stock is and haven't made an error in your request), this information doesn't add anything. However, if you are mistaken about the stock's value or misstate your request, the interactive agent can use the extra information to catch the error. Of course, these linguistic cues are not required—if you type "Tell me when IBM hits 90" or just "IBM 90," the interactive agent will infer the type of target, using the current price as the starting point.
- Although we could have allowed just one target to be added to the watchlist at a time, we have built recursive subpatterns allowing the interactive agent to recognize complex watchlist requests containing multiple stocks and multiple targets per stock. Among the requests recognized are IBM 90, IBM 70 90, MSFT GE 20 30, IBM 70 90 MSFT GE 20 30, "Tell me if IBM hits 70 or 90 and if MSFT or GE falls to 20 or climbs to 30," and so on.

You may feel that such functionality is overkill for this simple interactive agent. However, our experience has shown this type of conversational sophistication (or the lack thereof) can sometimes make or break an application's usability. In these cases, BuddyScript's high-level tools for handling conversational language make it comparatively simple to tackle problems that might otherwise be prohibitively difficult to solve.

Nuts and bolts

Our script ends with a handful of procedures and routines designed to handle common situations that arise when users interact with an interactive agent. Examples of these include requests for help, greetings and goodbyes, and user inputs that don't match any of the patterns in the script.

```
######################
# Help, Hello, etc. #
######################

procedure DisplayHelp
   - I can:
      Give you stock quotes {action=ExplainQuotes}
      Manage a portfolio of your favorite stocks {action=ExplainPortfolio}
      Notify you when stocks hit target values {action=ExplainWatchlist}
   do nothing
   - <blank/>
      >> Type "help" at any time to return to this menu...
   action ExplainQuotes
      - To get stock quotes, just type a list of one or more stock symbols. For example:
         IBM {*}
         IBM GM GE {*}
```



```
        do nothing
        - <blank/>
          To return to the main Help menu
          Help {*}
    action ExplainPortfolio
        - To add stocks to your portfolio, just type "add" followed by one or more symbols:
          Add IBM GM GE {*}
        do nothing
        - <blank/>
          To display your portfolio:
          My portfolio {*}
        do nothing
        - <blank/>
          To remove stocks, type "remove" followed by one or more symbols:
          Remove IBM GM GE {*}
        do nothing
        - <blank/>
          To clear your portfolio (remove all stocks), just type:
          Clear my portfolio {*}
        do nothing
        - <blank/>
          To return to the main Help menu
          Help {*}
    action ExplainWatchlist
        - To "watch" a stock (be notified when it reaches a particular target value):
          IBM 100 {*}
          IBM 80 100 {*}
          Tell me if IBM goes above 100 or MSFT falls below 20 {*}
        do nothing
        - <blank/>
          To return to the main Help menu
          Help {*}

procedure overrides ABFirstProc
    - Welcome, SYS.User.ScreenName!
        <blank/>
        Type "help" to see what I can do!
    exit all

procedure overrides ABHelloProc
    - Hello again, SYS.User.ScreenName!
        <blank/>

+ help
    call DisplayHelp

+ =Hello
    - Hi!

+ =Goodbye
    - Bye!

+ =Catch
    - I'm sorry, I don't understand. To see what I can do, type "help!"
```

Most of this code should be fairly self-explanatory, but a couple of points are worth discussing. First, the special system procedures ABFirstProc and ABHelloProc are automatically called when the user visits the interactive

agent for the first time, and when the user initiates each session with the interactive agent, respectively. There are default implementations for these procedures (and a host of other system procedures) in the BuddyScript libraries, but it often makes sense to override them in your script.

Finally, the subpattern Catch is also defined in the BuddyScript libraries. It matches absolutely anything, but assigns a very low score; consequently, the routine in which we refer to the Catch subpattern should match if and only if none of the other patterns in the script matches the current user input. Writing such a routine is a simple way to handle cases in which the user asks for something outside the interactive agent's realm of expertise.

Conclusion

We hope this article series has helped open your eyes to the potential of interactive agents deployed in Lotus Instant Messaging (and other types of messaging environments). For additional developer information and more examples, visit the BuddyScript home page where you'll find the latest BuddyScript software and a growing developer community.

ABOUT THE AUTHOR

Gray Norton is a Senior Product Manager for ActiveBuddy, where he oversees the company's BuddyScript product line. He has been in the software industry since the early 1990's and has played key product management roles for several companies in the interactive media space, including Electrifier, Motion Factory, MetaCreations, and Ray Dream. Gray is a graduate of Stanford University's Symbolic Systems Program.

