Easily translatable Shiny applications

Matjaž Jeran
(matjaz.jeran@bsi.si)
Bank of Slovenia, Ljubljana, Slovenia

ABSTRACT

The article deals with multicultural environments where software with more that one language should be implemented. It deals specifically with programs developed in R and Shiny. It shows programming techniques to use single program code for many target languages used for user interface.

Keywords*: programming, R, Shiny, multi-lingual applications, GNU gettext, DataTable*

JEL Classification codes*: Y - Miscellaneous Categories*

1. REASONS FOR MAKING EASILY TRANSLATABLE SOFTWARE

Software development has always been a difficult and expensive work. If the products are planned for use in a wide audience, the cost of development is distributed to as many customers as possible. The language of the user interface may be the limiting border where the software may be used. That is especially true for small language groups such as people speaking Slovenian.

Some countries have more than one official language in use such as Canada or Switzerland or in some parts of Slovenia. International organizations such as EU or UN also practice the use of more languages. In all such cases ability of applications to communicate in more than just one language is obligatory or at least desired.

Making software "speaking" more languages can also be purely business advantage. It can spread the software from the author territory to another natural language territory. If translation of the program user interface is easy, a tested piece of software can soon be ready to use in another territory. If there are many translations, the use of software can be shared within a group of countries or for international community such as countries reporting to Eurostat or members of European Union. [1]

2. ANY MODELS YET?

Linux community already uses GNU gettext library [2] that offers features to write easily translatable C programs. First versions of that library were made for Sun Solaris in the years soon after 1990. The project contains not only software library but also software tools to facilitate translation of strings from one language to another such as poedit and others.

The same idea was implemented later in other programming languages.

In the time when this article was almost finished, I noticed R package called "shiny.i18n" available on github. [3] According to the statement of the author(s) of this package, it is still in its infancy, and so the programming interface might be changed.

The base R package contains function gettext. It is used for the base R and core packages to communicate diagnostic messages in all languages that the translation was provided for. This article will try to demonstrate something similar for Shiny applications.

3. HOW TO MAKE SOFTWARE EASILY TRANSLATABLE

The basic principles of easily translatable software will be shown on the very simple program "Hello world". [4] The simple R code for this would be

```
print ("Hello world")
```

This simple program has two components of the program merged in a single line: the "print" **command** and the **message** "Hello world". In order to facilitate the translation, separation of the commands and the messages for user interface should be done as the following:

```
txt <- "Hello word"
print (txt)
```

This two-line program has the messages separated from the command part. All messages (in this case only one) are collected in the initial part of the program, all the commands are at the last part. The commands are not affected by changing the natural language by which the program communicates with its users.

All the messages used by the program can be stored in a vector. In the simple case above, the vector contains only one message - the "Hello world".

In order to have a dual language program, a two column matrix can be used instead of a vector and a selection object can be added to define which of the two languages is the chosen one. Such as matrix with English and Slovenian message is:

```
lang <- "en"
txt <- matrix (c ("Hello world", "Pozdravljen svet"),
 nrow = 1, ncol = 2,
 dimnames = list (c ("Hello world"), c ("en", "sl")))
print (txt ["Hello world", lang])
```

The object `lang` defines the language, the txt all the messages in all languages. The print command is invariant from the natural language used.

To add more languages, the matrix could be expanded from two columns to three and more. A matrix with French message added could be like this:

```
lang <- "fr"
txt <- matrix (c ("Hello world", "Pozdravljen svet", "Bonjour le monde"),
 nrow = 1, ncol = 3,
 dimnames = list (c ("Hello world"), c ("en", "sl", "fr")))
print (txt ["Hello world", lang])
```

By observing the general structure of the program

By adding translations of the messages the program commands remained the same, only the part with the messages `txt` has been altered.

Adding more natural languages and expanding matrices is possible, but the matrix structure soon becomes too big to maintain efficiently. It is practically impossible to have a single matrix to support all EU languages because very few people can translate message to more than 20 languages.

It is also inefficient to keep all translations in RAM during the execution of the program.

It is desirable to find 20 people to maintain pairwise translations from English to their native language. Instead of one matrix with many columns, the program can be organized to use many two-column matrices, each matrix having original language (e.g. English) and the translation. A two-column matrix could be made by a translator for each target language.

This schema can be used to organize a mass and parallel translation of original set of messages to many languages. The message files for various languages can be organized in a tree data file structure. The naming of the languages can be adopted from ISO standard ISO-639 [5]. Potential dialects

and other local customs can be solved by using variants denoted by countries designated by standard for country names and code elements ISO-3166-1 [6].

For example on Fedora 28 linux the English and German translation files for R can be found on

```
/lib64/R/library/translations/
                                ./en/LC_MESSAGES/R.mo
                                ./de/LC_MESSAGES/R.mo
```

The English, German and Slovenian translation files for package Rcmdr can be found on

```
/lib64/R/library/Rcmdr/po/
                            ./en/LC_MESSAGES/R.mo
                            ./de/LC_MESSAGES/R.mo
                            ./sl/LC_MESSAGES/R.mo
```

Similar structure is created by default on MS Windows. The start of the path is

```
C:\Program files\R\<version_of_R>\library.
```

The initialization of the messages matrix can be done by reading the text file containing all the messages. The contents of the Italian tab delimited translation file would be:

```
en     it
Hello world   Buon giorno il mondo
```

with the easy translatable R program:

```
lang <- "it"

txt <- read.table (file = file.path (lang, "mesg.txt"),
 header = TRUE, sep = "\t", stringsAsFactors = FALSE)
rownames (mesg) <- txt [, 1]

print (txt ["Hello world", lang])
```

The complete separation of messages from the program code causes the possibility of having the message file to be out of sync with the code. E.g. the code contains some new features with new messages like "Good evening", but the translation of those messages is not made yet. Potential referencing the non-existent message would result in program crash. The remedy for that is replacement of direct reference to messages matrix by a function that contains the error checking and diagnostics and so prevents the program to crash.

The function could be implemented like this:

```
gettxt <- function (m, lang, def.lang = "en")
# getting message text m with suitable diagnostics if string is
missing
# if translation is missing, get original (e.g English) string
{
  if (m %in% txt [, def.lang]) {
    tmp <- txt [m, ]
    if (is.na (tmp [lang]) | tmp [lang] == "") {
      tmp <- tmp [def.lang]
    } else tmp <- tmp [lang]
  } else
    tmp <- paste0 ("Missing string '", m, "' - check translation
file")
  return (tmp)
}
```

The proposed function and data structure is very similar to GNU gettext solution and could be easily adapted to gettext API if such API is available for Shiny in future.

Similar data structure is used by other programs in Linux community: such as R [7], R Commander [8], KDE [9], GNOME [10] etc. The key routine used is available in GNU gettext tools. The source translations are kept in .po files. The translation of the messages is supported by various utilities such as "po editors": poedit [11] and Lokalize [12]. In our case the messages are kept in tab delimited message file for each language.

4. OTHER HIDDEN DEPENDENCIES OF NATURAL LANGUAGES IN R

User interface is not the only dependency of R programs to natural languages. R is especially known for "programming with data". Data obtained from external file or databases can less evidently introduce dependencies to a program. Column names in a spreadsheet usually contains a name or abbreviation of a name in the natural language where data was collected. E. g. a price list in English can have columns: Quantity, Item and Price. The equivalent price list columns in French could be Quantité, Matière and Prix. Let us suppose that the order of the columns is the same for all languages.

4.1 Data structures and natural languages

If the files are read by

```
df <- read.table (..., header = TRUE)
```

these column names are implicitly imported in the program and are used when showing or printing the data.frame that is processed. These column names may be different in other natural language environment but the order of columns is the same. To avoid language dependency of the code, only positional referencing of the data.frame elements should be used. Data.frames should be treated the same way as matrices, e.g. by referencing the price column of data.frame df: instead of `df$Price` or `df$Prix` we should use df `[, 3]`.

To avoid explicit naming of the columns, reading the headers should be omitted:

```
df <- read.table (..., header = FALSE, skip = 1)
```

This reading will set default column names of the data.frame: V1, V2, ... Vn. These column naming can be overridden by the names defined by the translator.

When rendering data table on the screen, the function DT::renderDataTable inherits the column names from internal data structure. To display column names in the target language, the target language set of column names of data.frames and other data structures should be used in the application:

```
colnames (df) <- c ("target_language_name", ...)
e.g. for price list:
colnames (df) <- c (gettxt ("Quantity", lang), gettxt ("Item",
lang),
                    gettxt ("Price", lang))
```

This set of column names should be used to rename a copy of the data.frame to be displayed on the screen.

The same issue occurs to display a graph with default values. Fortunately, non-default values as optional parameters in functions producing graphs can be suplied, so renaming of data structures is not necessary.

This coding strategy is heavily dependent on the stability of data structures. By adding or omitting a column of the data.frame used, all code referring to column positions must be "renumbered". The execution of the code before adaptation to altered data structure may be completely wrong.

4.2 Data contents in natural languages

The data can contain numeric data. It is expected, that all cultures have same meaning for numbers. The data can also contain texts that are natural language dependent. The column Item or Matière contains English or French words describing the price list.

The only way to avoid the dependency is to use some code list with all translations. In this case the codes are used internally, the translations of the code lists are displayed depending on the target language. In our price list example: abstract code A may be used for water, wasser, voda, l'eau and code B for beer, bier, pivo, bière and so on. The English, German, Slovenian and French code list define the behavior of the program in these target environments.

5. OTHER CULTURAL DEPENDENCIES

The behavior of a program is defined by the variables that are already defined by the operating system environment. The environment is defined by the word done by other people, trying to adapt the computer system to target natural language and territory. The feature implicitly defined are the code-page - allowing the use of special characters not found in English language, collating sequence (used for lexical sort), using currency symbols, dates and calendars, displaying numbers.

Majority of European languages have the basic localizations on the operating system completed, so very few additional activities are required for Shiny applications.

Dates in R are usually displayed in ISO format yyyy-mm-ss [13] which can be used by any language, the decimal symbol can be defined by `options(OutDec = ",")` or `options(OutDec = ".")` Sometimes Shiny displays date or time values as numbers. In such cases we use the R routines to convert the dates to strings.

Shiny might use the R package DataTable or DT to display tables. Use of this package must be adapted with plug-ins for the target language, otherwise the internal user interface for DataTable would be English. Details about the use of plug-ins can be found in the manual for the package DataTable [14].

6. TRANSLATIONS OF SHINY APPLICATIONS

The Shiny applications have three distinct parts: the global part, the user interface and the server part. The language selector and the messages text are placed in the global part and are used in the user interface and in the server part.

The UI part contains the design of screens, the drop down menus and forms. All menus and form text are language dependent, so that all texts should be collected in the global part.

In the server part the Shiny code refers to global texts of messages and the data frames and tables using positional notation. In order to display target language column names, column names of the data.frame or table should be translated before calling renderDataTable.

7. SELECTION OF THE LANGUAGE USED

The selection of the language used with Shiny applications can be determined on the server e.g. by introductory web page with hyperlinks for every language provided or by the client. The browser settings usually contain a ordered list of preferred languages, some javascript code on the introductory web page could select the best language such navigator language property as proposed by w3schools [15], then the values obtained can be transferred to Shiny [16].

8. HOW TO CHANGE APPLICATION FROM SINGLE TO MULTI-LANGUAGE?

The procedure to modify already made Shiny application to a multi-language one consists of several steps. Each step from the list bellow should be finished by testing.

1. Insert definition of language selector at the beginning of application. Add txt matrix and identification of all messages in the application. Add each distinct message to txt matrix in the global part of the Shiny application.
2. Insert the function gettxt in the global part of Shiny application just before UI.
3. Modify each occurrence of message "m" in UI or server part to `gettxt ("m", lang)`. At the end of this stage, the complete user interface should be translated.
4. Select the data model strategy. Replace all references of column names of data.frames in the code by matrix like positional references. Rename all column names of data.frames and tables before using renderDataTable.
5. Completing texts for all translations and exhaustive testing of the application in all available languages. Check for variations of the text (cases, singular, plural, dual in Slovenian etc.)

9. CONCLUSIONS

The coding strategy was implemented on a couple of Shiny applications in the Bank of Slovenia: currency exchange, financial accounts, balance of payments for energy, real estate markets.

A demo was also made to show all the techniques discussed in the article and is available from the author.

Certain language behavior of Shiny applications are driven by system javascript libraries. In order to adapt behavior of these libraries, they must be adapted to target language as well.

The missing part for Slovenian version of the demo is package DT showing instructions in English. In order this part of user interface to in Slovenian, a Slovenian plug-in for DT library should be used.

This example shows that all R packages that issue some message should be written in easily translatable way and behaving according to natural language support settings.

The experience with real applications shows what improvement to simplify and facilitate programming multilingual applications could be made:

- allow to specify column names with calling function DT::renderDataTable
- easier change of language interface of DT::renderDataTable
- allow the possibility to use GNU gettext call with using user defined .po files. So far R base function gettext allows to use R internal messages [17]

References:

1. European Union (retrieved 06.09.2018) ***https://europa.eu/european-union/***
2. GNU Project gettext (retrieved 06.09.2018) ***https://www.gnu.org/software/gettext/***
3. shiny.i18n - Shiny applications internationalisation made easy! (retrieved 06.09.2018) ***https://appsilon.github.io/shiny.i18n/***
4. "Hello, World!" program (retrieved 06.09.2018) ***https://en.wikipedia.org/wiki/%22Hello,_World!%22_program***
5. ISO 639 - Codes for the Representation of Names of Languages (retrieved 06.09.2018) ***https://en.wikipedia.org/wiki/ISO_639***
6. ISO 3166-1 Country names and code elements (retrieved 06.09.2018) ***https://en.wikipedia.org/wiki/ISO_3166-1***
7. R - internals (retrieved 06.09.2018) ***https://cran.r-project.org/manuals.html***
8. Rcmdr (retrieved 06.09.2018) ***https://CRAN.R-project.org/package=Rcmdr***
9. KDE Community Home (retrieved 06.09.2018) ***https://www.kde.org/***
10. GNOME - An easy and elegant way to use your computer (retrieved 06.09.2018) ***https://www.gnome.org/***

11. POEDIT - Powerful and intuitive translations editor (retrieved 06.09.2018)
https://poedit.net/
12. Lokalize - Computer-Aided Translation System (retrieved 06.09.2018)
https://www.kde.org/applications/development/lokalize/
13. ISO 8601 - Data elements and interchange formats (retrieved 06.09.2018)
https://en.wikipedia.org/wiki/ISO_8601
14. Internationalisation plug-ins (retrieved 06.09.2018)
https://datatables.net/plug-ins/i18n/
15. Navigator language property (retrieved 06.09.2018)
https://www.w3schools.com/jsref/prop_nav_language.asp
16. Joe Cheng: Communicating with Shiny via JavaScript (retrieved 06.09.2018)
http://shiny.rstudio.com/articles/communicating-with-js.html
17. R package DT (retrieved 06.09.2018)
https://cran.r-project.org/package=DT

Details of a demo program

Appendix

If the messages of a bilingual program are kept in the program code, a part of the global part could look like this:

```
# the language selector
lang <- "en"

versionn <- "07.09.2018"

# the messages for menus, forms and column names
txt <- matrix (c (
  paste ("Simple translated demo ver:", versionn),
  paste ("Demo primer ver:", versionn),
  "Data", "Podatki",
  "Summary", "Izvlecek",
  "Table", "Tabela",
  "View first", "Pogled prvih",
  ....
"Date", "Datum",     # data columns
  "Item", "Postavka",
  "Amount", "Kolicina",
  "Value", "Vrednost"),
  nrow = 23, ncol = 2, byrow = TRUE,
  dimnames = list (c ("Simple_translated_demo",
                      "Data", "Summary",
                      "Table", "View_first",  ...
                     "Date", "Item", "Amount", "Value"),
                   c ("en", "sl")))
```

From all messages we can store column names

```
data.colnames <- c (gettxt ("Date", lang), gettxt ("Item", lang),
                    gettxt ("Amount", lang), gettxt ("Value",
lang))
```

The data reading part is also probably done in the global part. The simplest way of reading data is done by template `df <- read.table (..., header = FALSE, skip = 1)`. The data columns are named V1, ... Vn. We refer to the variables by df [, i] in the server code.

The input and output lists names are independent of the natural language. The variables and labels are used for communication between the UI part and the server part.

The UI part defined the screen layout and the menus e.g.:

```
ui <- shiny::shinyUI (
  navbarPage (
      title = gettxt (paste ("Simple translated demo ver:",
versionn), lang),
    selected = "M11",
    navbarMenu (
      title = gettxt ("Data", lang),
      tabPanel (title = gettxt ("Summary", lang), value = "M11",
                                          sidebarLayout     (

          sidebarPanel (htmlOutput ("T11") ),
          mainPanel (
            tableOutput ("T11Text") ) ) )
    ),
    navbarMenu (
      title = gettxt ("Table", lang),
      tabPanel (
        title = gettxt ("View first", lang), value = "M21",
                                          sidebarLayout     (

          sidebarPanel (htmlOutput ("T21") ),
          mainPanel (
            tableOutput("T21table") ) ) ),
      tabPanel (
        title = gettxt ("View all", lang), value = "M22",
                                          sidebarLayout     (

          sidebarPanel (htmlOutput ("T22"),
                        dateRangeInput (inputId = "dates22",
                                        label = gettxt ("Date",
lang),
                                          start = m.date, end =
M.date,
```

```
                                        min = m.date, max = M.date,
                                          separator = gettxt ("to",
lang))
          ),
          mainPanel (
            dataTableOutput ("T22table") ) ) ),
....
```

The server part contains code to display tables and graphs. Tables are displayed using DT::renderDataTable where column names of input data structure are displayed on the screen. Output of graphs is called by shiny::renderPlot where many optional parameters determine target language texts on the graph.

```
server <- function (input, output)
{
  output$T11 <- shiny::renderText (gettxt ("Summary", lang))

 output$T11Text <- shiny::renderTable (colnames = FALSE, summary
(dat))

  output$T21 <- shiny::renderText (gettxt ("View first", lang))

  output$T21table <- shiny::renderTable (
    {
      n <- 10
      tmp <- dat [1:n, ]
      tmp [, 1] <- format (tmp[, 1], '%Y-%m-%d')  # ISO format
      colnames (tmp) <- data.colnames
      return (tmp)
    }
  )

  output$T22 <- shiny::renderText (gettxt ("View all", lang))

  output$T22table <- DT::renderDataTable (
    rownames = FALSE,
    {
       tmp <- subset (dat, subset = input$dates22 [1] <= dat [,
1] &
                                  dat [, 1] <= input$dates22 [2])
      colnames (tmp) <- data.colnames
      return (tmp)
    }
  )
```

Drawing graphs is done by referring to column positions and using column names defined in the global part.

```
output$T32plot <- shiny::renderPlot (
    {
      tmp <- subset (dat, subset = input$dates32 [1] <= dat [,
1] &
                                  dat [, 1] <= input$dates32 [2])
      if (nrow (tmp) > 0) {
        ggplot2::ggplot (data = tmp,
         aes (x = tmp [, 1], y = tmp [, 4], colour = tmp [, 2])) +
          geom_point () + geom_line () +
          scale_x_date (labels = scales::date_format()) +
          expand_limits (y = 0) +
          xlab (data.colnames [1]) + ylab (data.colnames [4]) +
          labs (col = data.colnames [2]) +
          theme_bw(base_size = 14, base_family = "sans")
      }
    }
  )
```

The complete demo can be obtained from the author.