A Generic Registry Infrastructure for R

David Meyer

2009-02-17

1 Introduction

More and more, R packages are offering dynamic functionality, allowing users to extend a "repository" of initial features or data. For example, the **proxy** package (Meyer and Buchta, 2008) provides an enhanced `dist()` function for computing dissimilarity matrices, allowing to choose among several proximity measures stored in a registry. Each entry is composed of a small workhorse function and some meta data including, e.g., a character vector of aliases, literature references, the formula in plain text, a function to coerce between similarity and distance, and a type categorization (binary, metric, etc.). Users can add new proximity measures to the registry at run-time and immediately use them without recreating the package, specifying one of the aliases defined in the meta data. Similarly, the **relations** (Hornik and Meyer, 2008) and **CLUE** (Hornik, 2005, 2007) packages use simple registries internally to link some meta data to available functions, used by the high-level consensus ranking and cluster ensemble functions, respectively.

Such a registry, whether exposed to the user or not, is conceptually a small in-memory data base where entries with a common field structure are stored and retrieved and whose fields can be of mixed type. At first sight, a data frame seems to be the data structure of choice for an appropriate implementation. Unfortunately, data frames are inconvenient to use with factors, functions, or other recursive types such as lists due to automatic coercions taking place behind the scenes. In fact, a simpler, record-like structure such as a list with named components ("fields") appears more practical. Also, features known from "real" data bases such as compound keys, validity checking of new entries, and use of access rights are not available by default and need to be "reinvented" every time they are needed.

The **registry** package provides a simple mechanism for defining and manipulating user-extensible registry objects. A typical use case in the context of an R package could include the following steps:

1. Create one or more registry objects inside the package's namespace.
2. Insert entries to the registry.
3. Possibly, "seal" the entries and set access rights.
4. Possibly, export the registry object to the user level.
5. Browse and retrieve entries from the registry.

In the following, we explain these steps in more detail: first, how a registry can be set up; second, how entries can be added, modified and retrieved; and third, how a registry can be sealed and restricted through the definition of access rights.

2 Creating Registries

A registry basically is a container (implemented in R as an environment), along with some access functions. A new object of class `registry` can simply be created using the `registry()` function:

```
> library(registry)
> R <- registry()
> print(R)

An object of class 'registry' with no entry.
```

Optional parameters include the specification of an (additional) class for the created registry object and the individual entries, as well as the specification of some validity function checking new entries to be added to the registry.

In the following, we will use the example of a simple address book, whose entries include first and last name, address, age, home/cell phone number, and a business/private classification. Last and first name build the search key. Age is an optional integer in the range of 1 and 99. Additionally, at least one phone number should be added to the registry.

We start by creating two simple validity functions. The first one, to be specified at field level later on, checks a given age:

```
> checkAge <- function(x) stopifnot(is.na(x) || x > 0 && x < 100)
```

The second one, specified at registry level, checks whether a given registry entry (list of named components) contains at least one phone number:

```
> checkPhone <- function(x) stopifnot(!is.na(x$mobile) || !is.na(x$home))
```

Next, we create a registry of class `Addressbook` (inheriting from `registry`), containing entries of class `Address` and using the above validity function.

```
> R <- registry(registry_class = "Addressbook", entry_class = "Address",
+               validity_FUN = checkPhone)
```

The additional class for the registry allows, e.g., user-defined printing:

```
> print.Addressbook <-
+ function(x, ...) {
+     writeLines(sprintf("An address book with %i entries.\n", length(x)))
+     invisible(x)
+ }
> print(R)

An address book with 0 entries.
```

At this stage, we are ready to set up the field information. First and last names are mandatory character fields, uniquely identifying an entry (key fields). Lookups should work with partial completion, ignoring case:

```
> R$set_field("last", type = "character", is_key = TRUE, index_FUN = match_partial_ignorecase)
> R$set_field("first", type = "character", is_key = TRUE, index_FUN = match_partial_ignorecase)
```

The address is also character, but optional:

```
> R$set_field("address", type = "character")
```

At least one phone number (character) is required. This can be achieved by making them optional, and using the validity function specified at the registry level to check whether one of them is empty:

```
> R$set_field("mobile", type = "character")
> R$set_field("home", type = "character")
```

The age field is an optional integer with a defined range, checked by the field-level validity function:

```
> R$set_field("age", type = "integer", validity_FUN = checkAge)
```

Finally, the business/private category is defined by specifying the possible alternatives (`Business` is set as default):

```
> R$set_field("type", type = "character",
+             alternatives = c("Business", "Private"),
+             default = "Business")
```

The setup for a field can be retrieved using `get_field()`:

```
> R$get_field("type")
```

```
          type character
  alternatives c("Business", "Private")
       default Business
  is_mandatory FALSE
 is_modifiable TRUE
        is_key FALSE
     index_FUN function (lookup, entry, ...)  tolower(lookup) %in%
               tolower(entry)
index_FUN_args list()
```

`get_fields()` returns the complete list.

3 Using Registries

We now can start adding entries to the registry:

```
> R$set_entry(last = "Smith", first = "Mary", address = "Vienna",
+             home = "734 43 34", type = "Private", age = 44L)
> R$set_entry(last = "Smith", first = "Peter", address = "New York",
+             mobile = "878 78 87")
```

If all field values are specified, the field names can be omitted:

```
> R$set_entry("Myers", "John", "Washington", "52 32 34", "898 89 99",
+             33L, "Business")
```

Duplicate or invalid entries are not accepted:

```
> TRY <- function(expr) tryCatch(expr, error = print)
> TRY(R$set_entry(last = "Smith", first = "Mary"))
```

```
<simpleError: Entry already in registry.>
```

```
> TRY(R$set_entry(last = "Miller", first = "Henry"))
```

```
<simpleError: !is.na(x$mobile) || !is.na(x$home) is not TRUE>
```

```
> TRY(R$set_entry(last = "Miller", first = "Henry", age = 12.5))
```

```
<simpleError: Field "age" does not inherit from: integer>
```

```
> TRY(R$set_entry(last = "Miller", first = "Henry", age = 999L))
```

```
<simpleError: is.na(x) || x > 0 && x < 100 is not TRUE>
```

A single entry can be retrieved using `get_entry()`:

```
> R$get_entry(last = "Smith", first = "mar")
```

```
   last Smith
  first Mary
address Vienna
 mobile NA
   home 734 43 34
    age 44
   type Private
```

Since returned entries inherit from `Address`, we can provide a user-defined print method:

```
> print.Address <- function(x) with(x,
+     writeLines(sprintf("%s %s, %s; home: %s, mobile: %s; age: %i (%s)", first, last, address, h
> R$get_entry(last = "Smith", first = "mar")

Mary Smith, Vienna; home: 734 43 34, mobile: NA; age: 44 (Private)
```

Note that even though the first name of Mary Smith is incompletely specified and in lower case, the lookup is still successful because of the partial matching indexing function. The `[[` operator can be used as an alternative to `get_entry()`:

```
> R[["Myers"]]

John Myers, Washington; home: 898 89 99, mobile: 52 32 34; age: 33 (Business)
```

For Myers, the last name uniquely identifies the entry, so the first name can be omitted. Key values can have alternative values:

```
> R$set_entry(last = "Frears", first = c("Joe", "Jonathan"),
+             address = "Washington", home = "721 42 34")
```

Either of them can be used for retrieval:

```
> identical(R[["Frears", "Jonathan"]], R[["Frears", "Joe"]])

[1] TRUE
```

Unsuccessful lookups result in a return of `NULL`. Multiple entries can be retrieved using the `get_entries()` accessing function. They are returned in a list whose component names are generated from the key values:

```
> R$get_entries("Smith")

$Smith_Mary
Mary Smith, Vienna; home: 734 43 34, mobile: NA; age: 44 (Private)

$Smith_Peter
Peter Smith, New York; home: NA, mobile: 878 78 87; age: NA (Business)
```

Full-text search in all information is provided by `grep_entries()`:

```
> R$grep_entries("Priv")

$Smith_Mary
Mary Smith, Vienna; home: 734 43 34, mobile: NA; age: 44 (Private)
```

A list of all entries can be obtained using either of:

```
> R$get_entries()
> R[]
```

The summary method for registry objects returns a data frame:

```
> summary(R)
```

```
    last first    address    mobile      home age     type
1  Smith  Mary     Vienna      <NA> 734 43 34  44  Private
2  Smith Peter   New York 878 78 87      <NA>  NA Business
3  Myers  John Washington  52 32 34 898 89 99  33 Business
4 Frears   Joe Washington      <NA> 721 42 34  NA Business
```

Entries can also be modified using `modify_entry()`, specifying key and new field values:

```
> R[["Smith", "Peter"]]
```

```
Peter Smith, New York; home: NA, mobile: 878 78 87; age: NA (Business)
```

```
> R$modify_entry(last = "Smith", first = "Peter", age = 22L)
> R[["Smith", "Peter"]]
```

```
Peter Smith, New York; home: NA, mobile: 878 78 87; age: 22 (Business)
```

Finally, entries can be removed using `delete_entry()`:

```
> R$delete_entry(last = "Smith", first = "Peter")
> R[["Smith", "Peter"]]
```

```
NULL
```

4 Sealing Registries and Setting Access Rights

Occasionally, developers might want to protect a registry that ships with some package to prevent accidental deletions or alterations. For this, **registry** offers two mechanisms: first, a registry object can be "sealed" to prevent modifications of *existing* data:

```
> R$seal_entries()
> TRY(R$delete_entry("Smith", "Mary"))
```

```
<simpleError: Deletion of entry not allowed.>
```

```
> R$set_entry(last = "Slater", first = "Christian", address = "Boston",
+             mobile = "766 23 88")
> R[["Slater"]]
```

```
Christian Slater, Boston; home: NA, mobile: 766 23 88; age: NA (Business)
```

Second, the access permissions for registries can be restricted:

```
> R$get_permissions()
```

```
   set_entries modify_entries delete_entries     set_fields
          TRUE            TRUE            TRUE           TRUE
```

```
> R$restrict_permissions(delete_entries = FALSE)
> TRY(R$delete_entry("Slater"))
```

```
<simpleError: Deletion of entries not allowed.>
```

```
> R$modify_entry(last = "Slater", first = "Christian", age = 44L)
> R[["Slater"]]
```

```
Christian Slater, Boston; home: NA, mobile: 766 23 88; age: 44 (Business)
```

References

K. Hornik. A CLUE for CLUster Ensembles. *Journal of Statistical Software*, 14(12), September 2005. URL http://www.jstatsoft.org/v14/i12/.

K. Hornik. *clue: Cluster ensembles*, 2007. URL http://CRAN.R-project.org/. R package version 0.3-20.

K. Hornik and D. Meyer. *relations: Data Structures and Algorithms for Relations*, 2008. R package version 0.5.

D. Meyer and C. Buchta. *proxy: Distance and Similarity Measures*, 2008. R package version 0.4-1.