Package 'R6P'

January 20, 2025

Type Package

Title Design Patterns in R

URL https://tidylab.github.io/R6P/, https://github.com/tidylab/R6P

BugReports https://github.com/tidylab/R6P/issues

Version 0.4.0

Maintainer Harel Lustiger <tidylab@gmail.com>

Description Build robust and maintainable software with object-oriented design patterns in R. Design patterns abstract and present in neat, well-defined components and interfaces the experience of many software designers and architects over many years of solving similar problems. These are solutions that have withstood the test of time with respect to re-usability, flexibility, and maintainability. 'R6P' provides abstract base classes with examples for a few known design patterns. The patterns were selected by their applicability to analytic projects in R. Using these patterns in R projects have proven effective in dealing with the complexity that data-driven applications possess.

License MIT + file LICENSE

Encoding UTF-8

RoxygenNote 7.3.2

Language en-GB

Depends R (>= 4.1)

Suggests testthat, DBI, RSQLite, ggplot2

Imports collections, dplyr, stringr, R6, tibble, tidyr

Config/testthat/edition 3

NeedsCompilation no

Author Harel Lustiger [aut, cre] (<https://orcid.org/0000-0003-2953-9598>), Tidylab [cph, fnd]

Repository CRAN

Date/Publication 2024-12-22 09:40:06 UTC

Contents

NullObject 2
Repository 2
Singleton 5
ValueObject 6

Index **8**

`NullObject` *Null Object Pattern*

Description

Model a domain concept using natural lingo of the domain experts, such as "Passenger", "Address", and "Money".

Usage

```
NullObject()
```

See Also

Other base design patterns: `Singleton`, `ValueObject()`

Examples

```
# See more examples at <https://tidylab.github.io/R6P/articles>

colnames(NullObject())
nrow(NullObject())
```

`Repository` *Repository Pattern*

Description

Mediates between the domain and data mapping layers using a collection-like interface for accessing domain objects.

Super class

`R6P::Singleton` -> `Repository`

Methods

Public methods:

- AbstractRepository$new()
- AbstractRepository$add()
- AbstractRepository$del()
- AbstractRepository$get()

Method `new()`**:** Instantiate an object

Usage:

```
AbstractRepository$new()
```

Method `add()`**:** Add an element to the Repository.

Usage:

```
AbstractRepository$add(key, value)
```

Arguments:

`key` (`character`) Name of the element.

`value` (?) Value of the element. Note: The values in the `Repository` are not necessarily of the same type. That depends on the implementation of `AbstractRepository`.

Method `del()`**:** Delete an element from the Repository.

Usage:

```
AbstractRepository$del(key)
```

Arguments:

`key` (`character`) Name of the element.

Method `get()`**:** Retrieve an element from the Repository.

Usage:

```
AbstractRepository$get(key)
```

Arguments:

`key` (`character`) Name of the element.

Examples

```
# See more examples at <https://tidylab.github.io/R6P/articles>

# The following implementation is a Repository of car models with their
# specifications.

# First, we define the class constructor, initialize, to establish a
# transient data storage.

# In this case we use a dictionary from the collections package
# <https://randy3k.github.io/collections/reference/dict.html>

# Second, we define the add, del and get functions that operate on the dictionary.
```

```
# As an optional step, we define the NULL object. In this case, rather then
# the reserved word NULL, the NULL object is a data.frame with 0 rows and
# predefined column.

TransientRepository <- R6::R6Class(
  classname = "Repository", inherit = R6P::AbstractRepository, public = list(
    initialize = function() {
      private$cars <- collections::dict()
    },
    add = function(key, value) {
      private$cars$set(key, value)
      invisible(self)
    },
    del = function(key) {
      private$cars$remove(key)
      invisible(self)
    },
    get = function(key) {
      return(private$cars$get(key, default = private$NULL_car))
    }
  ), private = list(
    NULL_car = cbind(uid = NA_character_, datasets::mtcars)[0, ],
    cars = NULL
  )
)

# Adding customised operations is also possible via the R6 set function.
# The following example, adds a query that returns all the objects in the database

TransientRepository$set("public", "get_all_cars", overwrite = TRUE, function() {
  result <- private$cars$values() |> dplyr::bind_rows()
  if (nrow(result) == 0) {
    return(private$NULL_car)
  } else {
    return(result)
  }
})

# In this example, we use the mtcars dataset with a uid column that uniquely
# identifies the different cars in the Repository:
mtcars <- datasets::mtcars |> tibble::rownames_to_column("uid")
head(mtcars, 2)

# Here is how the caller uses the Repository:

## Instantiate a repository object
repository <- TransientRepository$new()

## Add two different cars specification to the repository
repository$add(key = "Mazda RX4", value = dplyr::filter(mtcars, uid == "Mazda RX4"))
repository$add(key = "Mazda RX4 Wag", value = dplyr::filter(mtcars, uid == "Mazda RX4 Wag"))
```

```
## Get "Mazda RX4" specification
repository$get(key = "Mazda RX4")

## Get all the specifications in the repository
repository$get_all_cars()

## Delete "Mazda RX4" specification
repository$del(key = "Mazda RX4")

## Get "Mazda RX4" specification
repository$get(key = "Mazda RX4")
```

`Singleton` *Singleton*

Description

Enforces a single instance of a class and provides a global access point.

Details

This is an abstract base class. Instantiating `Singleton` directly triggers an error. Classes inheriting from `Singleton` share a single instance.

Methods

Public methods:

- `Singleton$new()`

Method `new()`**:** Create or retrieve an object

Usage:

```
Singleton$new()
```

See Also

Other base design patterns: `NullObject()`, `ValueObject()`

Examples

```
# See more examples at <https://tidylab.github.io/R6P/articles>
address <- function(x) sub("<environment: (.*)>", "\\1", capture.output(x))

# In this example we implement a `Counter` that inherits the qualities of Singleton
Counter <- R6::R6Class("Counter", inherit = R6P::Singleton, public = list(
  count = 0,
  add_1 = function() {
    self$count <- self$count + 1
    invisible(self)
  }
```

```
))

# Whenever we call the constructor on `Counter`, we always get the exact same instance:
counter_A <- Counter$new()
counter_B <- Counter$new()

identical(counter_A, counter_B, ignore.environment = FALSE)

# The two objects are equal and located at the same address; thus, they are the same object.

# When we make a change in any of the class instances, the rest are changed as well.

# How many times has the counter been increased?
counter_A$count

# Increase the counter by 1
counter_A$add_1()

# How many times have the counters been increased?
counter_A$count
counter_B$count
```

`ValueObject` *Value Object Pattern*

Description

Model a domain concept using natural lingo of domain experts, such as "Passenger," "Address," or "Money."

Usage

```
ValueObject(given = NA_character_, family = NA_character_)
```

Arguments

`given`	(`character`) A character vector with the given name.
`family`	(`character`) A character vector with the family name.

See Also

Other base design patterns: `NullObject()`, `Singleton`

Examples

```
# See more examples at <https://tidylab.github.io/R6P/articles>

# In this example we are appointing elected officials to random ministries, just
# like in real-life.
Person <- ValueObject
Person()

# Create a test for objects of type Person
# * Extract the column names of Person by using its Null Object (returned by Person())
# * Check that the input argument has all the columns that a Person has
is.Person <- function(x) all(colnames(x) %in% colnames(Person()))

# A 'Minister' is a 'Person' with a ministry title. The Minister constructor
# requires two inputs:
# 1. (`Person`) Members of parliament
# 2. (`character`) Ministry titles
Minister <- function(member = Person(), title = NA_character_) {
  stopifnot(is.Person(member), is.character(title))
  stopifnot(nrow(member) == length(title) | all(is.na(title)))

  member |> dplyr::mutate(title = title)
}

# Given one or more parliament members
# When appoint_random_ministries is called
# Then the parliament members are appointed to an office.
appoint_random_ministries <- function(member = Person()) {
  positions <- c(
    "Arts, Culture and Heritage", "Finance", "Corrections", "Racing",
    "Sport and Recreation", "Housing", "Energy and Resources", "Education",
    "Public Service", "Disability Issues", "Environment", "Justice",
    "Immigration", "Defence", "Internal Affairs", "Transport"
  )

  Minister(member = member, title = sample(positions, size = nrow(member)))
}

# Listing New Zealand elected officials in 2020, we instantiate a Person object,
# appoint them to random offices, and return a Minister value object.
set.seed(2020)
parliament_members <- Person(
  given = c("Jacinda", "Grant", "Kelvin", "Megan", "Chris", "Carmel"),
  family = c("Ardern", "Robertson", "Davis", "Woods", "Hipkins", "Sepuloni")
)

parliament_members
appoint_random_ministries(member = parliament_members)
```

Index

∗ **base design patterns**
NullObject, 2
Singleton, 5
ValueObject, 6
∗ **object-relational patterns**
Repository, 2

AbstractRepository *(*Repository*)*, 2

NullObject, 2, *5, 6*

R6P::Singleton, *2*
Repository, 2

Singleton, *2*, 5, *6*

ValueObject, *2*, *5*, 6