Package 'R6P'

February 1, 2021

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 0.2.0

Date 2021-01-01

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

LazyData true

RoxygenNote 7.1.1

Language en-GB

Depends R (>= 3.5)

Suggests testthat, collections, dplyr, DBI, RSQLite, ggplot2, tibble, tidyr, pryr

Imports R6, purrr, stringr

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 2021-02-01 06:00:02 UTC

R topics documented:

`NullObject` *Null Object Pattern*

Description

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

Usage

```
NullObject()
```

Details

Caution: NullObject is designed for demonstration purposes. Instead of directly using the design pattern as it appears in the package, you'd have to adjust the source code to the problem you are trying to solve.

Null Object provides special behaviour for particular cases.

Note: The Null Object is not the same as the reserved word in R `NULL` (all caps).

How It Works:

When a function fails in R, some functions produce a run-time error while others return `NULL` (and potentially prompt a warning). What the function evokes in case of a failure is subjected to its programmer discretion. Usually, the programmer follows either a punitive or forgiving policy regarding how run-time errors should be handled.

In other occasions, `NULL` is often the result of unavailable data. This could happened when querying a data source matches no entries, or when the system is waiting for user input (mainly in Shiny).

If it is possible for a function to return `NULL` rather than an error, then it is important to surround it with null test code, e.g. `if(is.null(...)) do_the_right_thing()`. This way the software would do the right thing if a null is present.

Often the right thing is the same in many contexts, so you end up writing similar code in lots of places—committing the sin of code duplication.

Instead of returning `NULL`, or some odd value such as `NaN` or `logical(0)`, return a **Null Object** that has the same interface as what the caller expects. In R, this often means returning a `data.frame` structure, i.e. column names and variables types, with no rows.

When to Use It:

- In situations when a subroutine is likely to fail, such as loss of Internet or database connectivity. Instead of prompting a run-time error, you could return the **Null Object** as part of a gracefully failing strategy. A common strategy employs `tryCatch` that returns the **Null Object** in the case of an error:

```
# Simulate a database that is 5% likely to fail
read_mtcars <- function() if(runif(1) < 0.05) stop() else return(mtcars)

# mtcars null object constructor
NullCar <- function() mtcars[0,]

# How does the null car object look like?
NullCar()
#>  [1] mpg  cyl  disp hp   drat wt   qsec vs   am   gear carb
#> <0 rows> (or 0-length row.names)

# Subroutine with gracefully failing strategy
set.seed(1814)
cars <- tryCatch(
  # Try reading the mtcars dataset
  read_mtcars(),
  # If there is an error, return the Null Car object
  error = function(e) return(NullCar())
)

# Notice: Whether the subroutine fails or succeeds, it returns a tibble with
# the same structure.
colnames(cars)
#> [1] "mpg"  "cyl"  "disp" "hp"   "drat" "wt"   "qsec" "vs"   "am"   "gear" "carb"
```

- In Shiny dashboards

```
geom_null <- function(...){
  ggplot2::ggplot() + ggplot2::geom_blank() + ggplot2::theme_void()
}

if(exists("user_input")){
 ggplot2::ggplot(user_input, ggplot::aes(x = mpg, y = hp)) + ggplot2::geom_point()
} else {
  geom_null() + geom_text(aes(0,0), label = "choose an entry from the list")
}
```

- In unit-tests

```
classes <- function(x) sapply(x, class)
```

```
test_that("mtcars follows a certain table structure", {
    # Compare column names
    expect_identical(colnames(mtcars), colnames(NullCar()))
    # Compare variable types
    expect_identical(classes(mtcars), classes(NullCar()))
})
```

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.

Details

With **Repository**, in-memory objects do not need to know whether there is a database present or absent, they need no SQL interface code, and certainly no knowledge of the database schema.

How It Works:

- **Repository** isolates domain objects from details of the database access code;
- **Repository** concentrates code of query construction; and
- **Repository** helps to minimize duplicate query logic.

In R, the simplest form of **Repository** encapsulates `data.frame` entries persisted in a data store and the operations performed over them, providing a more object-oriented view of the persistence layer. From the caller point of view, the location (locally or remotely), the technology and the interface of the data store are obscured.

When to Use It:

- In situations with multiple data sources.
- In situations where the real data store, the one that is used in production, is remote. This allows you to implement a **Repository** mock with identical queries that runs locally. Then, the mock could be used during development and testing. The mock itself may comprise a sample of the real data store or just fake data.

- In situations where the real data store doesn't exist. Implementing a mock **Repository** allows you to defer immature decisions about the database technology and/or defer its deployment. In this way, the temporary solution allows you to focus the development effort on the core functionality of the application.
- In situations where using SQL queries can be represented by meaningful names. For example Repository$get_efficient_cars() = SELECT * FROM mtcars WHERE mpg > 20
- When building stateless microservices.

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 Pattern*

Description

Ensure a class only has one instance, and provide a global point of access to it.

Details

Singleton ensures a class only has one instance, and provide a global point of access to it.

How It Works:

1. Create only one instance of the **Singleton** class; and
2. If an instance exists, then serve the same object again.

The main features of **Singleton** are:

- Ensuring that one and only one object of the class gets created;
- Providing an access point for an object that is global to the program; and
- Controlling concurrent access to resources that are shared.

When to Use It:

- In situations that require exactly one instance of a class, that must be accessible to clients from a well-known access point. See the `Counter` example.

Caution: Singletons can be a problem in multi-threaded applications, especially when they manipulate mutable data.

Tip: Singletons work well for immutable data, such as reading from some data source, since anything that can't change isn't going to run into thread clash problems.

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 <- pryr::address

# 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()

sprintf("counter_A was crated at %s", address(counter_A))
sprintf("counter_B was crated at %s", address(counter_B))

identical(counter_A, counter_B)

# 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 of the
# instances 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 the domain experts, such as “Passenger”, “Address”, and “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.

Details

Caution: ValueObject is designed for demonstration purposes. Instead of directly using the design pattern as it appears in the package, you'd have to adjust the source code to the problem you are trying to solve.

A **Value Object** models a domain concept using natural lingo of the domain experts, such as "Passenger", "Address", and "Money".

Any **Value Object** is created by a function that receives input, applies some transformations, and outputs the results in some data structure such as a vector, a list or a data.frame.

How It Works:

In R, a good option for creating a **Value Object** is to follow two instructions:

- A **Value Object** is created by a `function`, rather than a class method; and
- A **Value Object** returns a `tibble`, rather than a list or a vector.

In essence, a **Value Object** is a data type, like `integer`, `logical`, `Date` or `data.frame` data types to name a few. While the built-in data types in R fit any application, **Value Objects** are domain specific and as such, they fit only to a specific application. This is because, `integer` is an abstract that represent whole numbers. This abstract is useful in any application. However, a **Value Object** represent a high-level abstraction that appears in a particular domain.

An example of a **Value Object** is the notion of a "Person". Any person in the world has a name. Needless to say, a person name is spelt by letters, rather than numbers. A **Value Object** captures these attribute as `tibble` columns and type checks:

```
Person <- function(given = NA_character_, family = NA_character_){
  stopifnot(is.character(given), is.character(family))
  stopifnot(length(given) == length(family))

  return(
    tibble::tibble(given = given, family = family)
    %>% tidyr::drop_na(given)
  )
}
```

Instantiating a person **Value Object** is done by calling the `Person` constructor function:

```
person <- Person(given = "Bilbo", family = "Baggins")
```

Getting to know the advantages of a **Value Object**, we should consider the typical alternative – constructing a Person by using the `tibble` function directly:

```
person <- tibble::tibble(given = "Bilbo", family = "Baggins")
```

Both implementations return objects with identical content and structure, that is, their column names, column types and cell values are identical. Then, why would one prefer using a **Value Object** and its constructor over the direct alternative?

There are four predominant qualities offered by the **Value Object** pattern which are not offered by the alternative:

1. Readability. Each **Value Object** captures a concept belonging to the problem domain. Rather than trying to infer what a `tibble` is by looking at its low-level details, the **Value Object** constructor descries a context on a high-level.
2. Explicitness. Since the constructor of the **Value Object** is a function, its expected input arguments and their type can be detailed in a helper file. Moreover, assigning input arguments with default values of specific type, such as `NA` (logical NA), `NA_integer_`, `NA_character_`, or `NA_Date` (see `lubridate::NA_Date`), expresses clearly the variable types of the **Value Object**.
3. Coherence. The representation of a **Value Object** is concentrated in one place – its constructor. Any change, mainly modifications and extensions, applied to the constructor promise the change would propagate to all instances of the Value Objects. That means, no structure discrepancies between instances that are supposed to represent the same concept.
4. Safety. The constructor may start with defensive programming to ensure the qualities of its input. One important assertion is type checking. Type checking eliminated the risk of implicit type coercing. Another important assertion is checking if the lengths of the input arguments meet some criteria, say all inputs are of the same length, or more restrictively, all inputs are scalars. Having a set of checks makes the code base more robust. This is because **Value Objects** are regularly created with the output of other functions calls, having a set of checks serves as pseudo-tests of these functions output throughout the code.

In addition to these qualities, there are two desirable behaviours which are not offered by directly calling `tibble`:

1. Null Value Object. Calling the **Value Object** constructor with no input arguments returns the structure of the `tibble` (column names and column types).
2. Default values for missing input arguments. In this manner, the **Value Object** has a well-defined behaviour for a person without a family name, such as Madonna and Bono.

In addition to native R data types, a **Value Object** constructor can receive other **Value Objects** as input arguments. Here are two examples that transmute Person to other Person-based concepts:

```
# A Passenger is a Person with a flight booking reference
Passenger <- function(person = Person(), booking_reference = NA_character_){
  stopifnot(all(colnames(person) %in% colnames(Person())))
  stopifnot(is.character(booking_reference))

  return(
    person
    %>% tibble::add_column(booking_reference = booking_reference)
    %>% tidyr::drop_na(booking_reference)
  )
}

person <- Person(given = "Bilbo", family = "Baggins")
passenger <- Passenger(person = person, booking_reference = "B662HR")
```

```
print(passenger)
#> # A tibble: 1 x 3
#>   given family  booking_reference
#>   <chr> <chr>   <chr>
#> 1 Bilbo Baggins B662HR

# A Diner is a Person that may have dinner reservation
Diner <- function(person = Person(), reservation_time = NA_POSIXct_){
  stopifnot(all(colnames(person) %in% colnames(Person())))
  stopifnot(is.POSIXct(reservation_time))

  return(
    person
    %>% tibble::add_column(reservation_time = reservation_time)
  )
}

person <- Person(given = "Bilbo", family = "Baggins")
timestamp <- as.POSIXct("2021-01-23 18:00:00 NZDT")
diner <- Diner(person = person, reservation_time = timestamp)
print(diner)
#> # A tibble: 1 x 3
#>   given family  reservation_time
#>   <chr> <chr>   <dttm>
#> 1 Bilbo Baggins 2021-01-23 18:00:00
```

When to Use It:

- In situations where domain concepts are more important then the database schema. For example, when you are modelling Passengers, your first instinct might be to think about the different data sources you'd need for the analysis. You may envision "FlightDetails" and "CustomerDetails". Next you will define the relationship between them. Instead, let the domain drive the design. Create a Passenger **Value Object** with the attributes you must have, regardless of any particular database schema.
- In a function that runs within a specific context. Rather than having an input argument called `data` of type `data.frame`, use the appropriate **Value Object** name and pass it its constructor.

```
Audience <- Person

## Without a Value Object
clean_audience_data <- function(data)
  dplyr::mutate(.data = data, given = stringr::str_to_title(given))

## With a Value Object
clean_audience_data <- function(attendees = Audience())
  dplyr::mutate(.data = attendees, given = stringr::str_to_title(given))
```

- In pipes and filters architecture.

Note: Value Objects do not need to have unit-tests. This is because of two reasons: (1) **Value Objects** are often called by other functions that are being tested. That means, **Value Objects** are

implicitly tested. (2) **Value Objects** are data types similarly to 'data.frame' or 'list'. As such, they need no testing

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. We capture that information
# in a new Value Object named 'Minister'.
# 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 Member 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