Let’s see how **OP** helps in defining function with semantic naming for parameter names, and in providing expected function return type. This will make use of many tools provided by package **wyz.code.offensiveProgramming**.

**Focus on a given type suffix**

Let’s consider you want to know more about type suffix *’s’*. Here is a typical session you could use.

library('wyz.code.offensiveProgramming')

# get the default type factory

tf <- retrieveFactory()

# is 's' a recorded type suffix?

tf$checkSuffix('s')

## [1] TRUE

# what means 's' type suffix

tf$getType('s')

## [1] "string"

# get verification function for type suffix 's'

tf$getVerificationFunction('s')

## function (o\_1\_)

## {

## if (!is.character(o\_1\_))

## return(FALSE)

## if (length(o\_1\_) == 0)

## return(TRUE)

## all([is.na](http://is.na)(o\_1\_) == FALSE)

## }

##

##

# get type factory information for 'pi'

tf$getRecordedTypes()[suffix == 's']

## suffix type verify\_function category

## 1: s string basic

**A real use case**

**Raw approach**

Let’s consider following function that helps organizing seats for your friend around the table *(under no constraint indeed)*.

organizeTable <- function(guestNames\_s) {

guestNames\_s[order(runif(length(guestNames\_s)))]

}

guests <- c('Marie', 'Yves-Marie', 'Fabien', 'Marcello', 'Tina')

organizeTable(guests)

## [1] "Marcello" "Marie" "Tina" "Yves-Marie" "Fabien"

organizeTable(character(0))

## character(0)

**Are those calls legal?**

From a R perspective, obvuously yes. From **OP** perspective, answer is more subtle and depends on evaluation modes you choose.

**Evaluation modes**

Currently, 3 evaluation modes are supported. They are incremental modes.

1. *standard\_R\_evaluation* mode,
2. *enhanced\_R\_evaluation* mode, adds function return type evaluation to previous mode,
3. *type\_checking\_enforcement* mode, adds function parameter type verification to previous mode.

# Available evaluation modes

defineEvaluationModes()

## [1] "standard\_R\_evaluation" "enhanced\_R\_evaluation"

## [3] "type\_checking\_enforcement"

**Standard R evaluation mode**

em <- EvaluationMode('standard\_R\_evaluation')

efrt <- 'x\_s' # expected function returned type

runTransientFunction(organizeTable, list(guests), em, efrt)

## $status

## [1] TRUE

##

## $value

## [1] "Tina" "Yves-Marie" "Marcello" "Fabien" "Marie"

##

## $mode

## [1] "standard\_R\_evaluation"

Using, *standard\_R\_evaluation* mode brings **status**, **value** and **mode** results. It does not provide any extraneous information than a call in your R console. This mode is provided to ease comparisons. Not to be used solely, indeed.

**Enhanced evaluation mode**

This mode checks function returned value against expected function return type.

em <- EvaluationMode('enhanced\_R\_evaluation')

runTransientFunction(organizeTable, list(guests), em, efrt)

## $status

## [1] TRUE

##

## $value

## [1] "Tina" "Marcello" "Marie" "Fabien" "Yves-Marie"

##

## $mode

## [1] "enhanced\_R\_evaluation"

##

## $function\_return\_type\_check

## parameter\_name parameter\_value validity

## 1: x\_s Tina,Marcello,Marie,Fabien,Yves-Marie TRUE

## message

## 1: good type in values

Using, *enhanced\_R\_evaluation* mode brings **function\_return\_type** results, as a *data.table*. The *validity* column provides information about type concordance between resulting value and expected type.

**Enhanced evaluation mode**

This mode checks parameter values against parameter semantic name specifications.

em <- EvaluationMode('type\_checking\_enforcement')

runTransientFunction(organizeTable, list(guests), em, efrt)

## $status

## [1] TRUE

##

## $value

## [1] "Yves-Marie" "Tina" "Marcello" "Fabien" "Marie"

##

## $mode

## [1] "type\_checking\_enforcement"

##

## $parameter\_type\_checks

## parameter\_name parameter\_value validity

## 1: guestNames\_s Marie,Yves-Marie,Fabien,Marcello,Tina TRUE

## message

## 1: good type in values

##

## $function\_return\_type\_check

## parameter\_name parameter\_value validity

## 1: x\_s Yves-Marie,Tina,Marcello,Fabien,Marie TRUE

## message

## 1: good type in values

Using, *type\_checking\_enforcement* mode brings **parameter\_type\_checks** results, as a *data.table*. Here also, the *validity* column provides information about type concordance between provided parameter value and parameter semantic name specification. Detail level is one line for each function parameter, to ease discovery of and remediance to uncompliances.

**So, legal or not ?**

Let’s check them

em <- EvaluationMode('type\_checking\_enforcement')

runTransientFunction(organizeTable, list(guests), em, efrt)

## $status

## [1] TRUE

##

## $value

## [1] "Tina" "Fabien" "Yves-Marie" "Marcello" "Marie"

##

## $mode

## [1] "type\_checking\_enforcement"

##

## $parameter\_type\_checks

## parameter\_name parameter\_value validity

## 1: guestNames\_s Marie,Yves-Marie,Fabien,Marcello,Tina TRUE

## message

## 1: good type in values

##

## $function\_return\_type\_check

## parameter\_name parameter\_value validity

## 1: x\_s Tina,Fabien,Yves-Marie,Marcello,Marie TRUE

## message

## 1: good type in values

runTransientFunction(organizeTable, list(character(0)), em, efrt)

## $status

## [1] TRUE

##

## $value

## character(0)

##

## $mode

## [1] "type\_checking\_enforcement"

##

## $parameter\_type\_checks

## parameter\_name parameter\_value validity message

## 1: guestNames\_s TRUE good type in values

##

## $function\_return\_type\_check

## parameter\_name parameter\_value validity message

## 1: x\_s TRUE good type in values

From R point of view, both calls are legal. From offensive programming, only calls returning a status **TRUE** are valid.

Accepting or not the second case is indeed a matter of input scope specification. Does organizing a table for no person have any sense?

If your answer is yes, then keep this function definition. If no, how could you improve previous implementation?

**Refined approach**

We will change the parameter name and specify length constraint on it. Note that function body is semantically exactly the same as for previous function. No change in implementation algorithm. Just parameter renaming propagation changes are applied.

organizeTableBis <- function(guestNames\_s\_1m) {

guestNames\_s\_1m[order(runif(length(guestNames\_s\_1m)))]

}

organizeTableBis(guests)

## [1] "Fabien" "Tina" "Marie" "Yves-Marie" "Marcello"

organizeTableBis(character(0))

## character(0)

runTransientFunction(organizeTableBis, list(guests), em, efrt)

## $status

## [1] TRUE

##

## $value

## [1] "Tina" "Fabien" "Marcello" "Yves-Marie" "Marie"

##

## $mode

## [1] "type\_checking\_enforcement"

##

## $parameter\_type\_checks

## parameter\_name parameter\_value validity

## 1: guestNames\_s\_1m Marie,Yves-Marie,Fabien,Marcello,Tina TRUE

## message

## 1: good type in values

##

## $function\_return\_type\_check

## parameter\_name parameter\_value validity

## 1: x\_s Tina,Fabien,Marcello,Yves-Marie,Marie TRUE

## message

## 1: good type in values

runTransientFunction(organizeTableBis, list(character(0)), em, efrt)

## $status

## [1] FALSE

##

## $value

## character(0)

##

## $mode

## [1] "type\_checking\_enforcement"

##

## $parameter\_type\_checks

## parameter\_name parameter\_value validity

## 1: guestNames\_s\_1m FALSE

## message

## 1: wrong length, was expecting [1m] , got [0]

##

## $function\_return\_type\_check

## parameter\_name parameter\_value validity message

## 1: x\_s TRUE good type in values

Now the second case is flagged as invalid from OP point of view. One of the reasons is given explicitly by the *parameter\_type\_check* *data.table*.

**To conclude …**

You have

1. been introduced to semantic naming for function parameter names,
2. been sensitized to semantic naming length specification impact,
3. been shown how and when to use predefined evaluation modes,
4. been initiated to use common **OP** tools, *runTransientFunction*, *EvaluationMode*, and function parameter type factory operations,
5. been trained to interpret evaluation results

Great, we are more than half the way. Next post will be about registering your own types, and managing your classes and their related functional tests.