Package ‘commandr’

February 19, 2015

Title Command pattern in R

Version 1.0.1

Author Michael Lawrence

Depends methods

Imports utils

Description An S4 representation of the Command design pattern. The Operation class is a simple implementation using closures and supports forward and reverse (undo) evaluation. The more complicated Protocol framework represents each type of command (or analytical protocol) by a formal S4 class. Commands may be grouped and consecutively executed using the Pipeline class. Example use cases include logging, do/undo, analysis pipelines, GUI actions, parallel processing, etc.

Maintainer Tengfei Yin <yintengfei@gmail.com>

License Artistic-2.0

NeedsCompilation no

Repository CRAN

Date/Publication 2014-08-25 20:22:16

R topics documented:

`Command-class` *Virtual Command Class*

Description

`Command` represents any generic operation and is often combined with other such objects in a sequence. This is the foundation for implementations of pipelines, logging, undo stacks, etc, according to the Command design pattern. Since `Command` is virtual, it cannot be constructed directly. Rather, one should construct an instance of a concrete derivative, like `Operation` or `Protocol`.

Details

Developers are encouraged to extend `Command` for new implementations of the Command design pattern.

There are a number of generics for which a `Command` derivative should provide methods:

`displayName(object)`: Gets the display name of the command, i.e., the name displayed in a user interface. The default implementation returns the class name. A `Command` may provide other visual attributes; the display name is considered fundamental for integration with user interfaces.

`widget(object, ...)`: Creates a widget for viewing and controlling this object. No default implementation.

`rev(object)`: Returns a `Command` that performs the opposite action, i.e., to undo an operation. This will not always be possible, so it is acceptable for a subclass to leave this unimplemented.

`active(object)`: Gets whether the command is considered active. This is meant for temporarily disabling or enabling a command without removing it completely from, e.g., a pipeline or GUI menu. No default implementation.

Of course, a `Command` implementation also needs a method that executes the command. The signature for such a function highly depends on the nature of the command, so the generic depends on the class.

Author(s)

Michael Lawrence

`Operation-class` *Operation: A Simple Command Implementation*

Description

An `Operation` object implements a `Command` with an R closure. This may be useful as a base for customized `Command` derivatives. Otherwise, consider it a skeletal proof-of-concept.

Constructors

`Operation(do, undo = NULL)`: Constructs an `Operation` that invokes the closure do. If undo is specified, it is the closure invoked for the reverse operation.

`OperationQ(do, undo)`: Constructs an `Operation` that invokes a function with the body taken from the quoted argument do. The function is enclosed in the calling environment. If undo is specified, it is treated equivalently to do, except it should implement the reverse operation.

Methods

`eval(expr)`: Executes the operation by evaluating the closure.

`rev(x)`: Returns a new `Operation` that performs the reverse operation, as long as undo was provided.

Author(s)

Michael Lawrence

Examples

```
op <- OperationQ(message("hello world"))
eval(op)

x <- 2
op <- OperationQ(x^2, sqrt(x))
x <- eval(op)
rop <- rev(op)
identical(2, eval(rop))
```

`Pipeline-class` *Pipeline: A Sequence of Protocols*

Description

A `Pipeline` represents a sequence of `Protocol`s. When executed, it executes each of the protocols in turn, where each protocol is passed the output of the previous protocol. `Pipeline` extends `list`, so, in general, it can be treated as one when it comes to subsetting, etc.

Constructor

`Pipeline(..., displayName = NULL)`: Constructs a `Pipeline` object with the protocols named in `...`. The arguments in `...` should be named and be either a `Protocol` object or a role name for which a default `Protocol` is constructed. Optionally, a user-readable display name can be specified as `displayName`.

Execution

`perform(object, data, ...)`: Executes the protocols in order, with the output of each pipeline passed as input to the next. Takes `data` as input to the first protocol and returns the output of the last protocol.

Accessors

`inType(object)`: Get the input type of the first protocol in the pipeline, or `NULL` if there are no protocols.

`outType(object)`: Get the input type of the first protocol in the pipeline, or `NULL` if there are no protocols.

`parameters(object)`: Obtains a list, each element of which is a list of the parameters for the corresponding protocol.

`protocol(object, role, method = character())`: Gets the first protocol with the matching `role` and `method`.

`protocol(object, role) <- value`: Replace the first protocol with the role given by `role` with `value`.

Subsetting

`pipeline(object, intype = "ANY", outtype = "ANY")`: Gets the sub-pipeline that spans from the first protocol with the input type derived from `intype` and the last protocol with the output type derived from `outtype`.

`head(x, n = 6L, role, method = character(), outtype)`: Like ordinary `head`, takes a prefix of the pipeline. If `outtype` is provided, this returns the pipeline through the first protocol with `outtype` as its output type. Otherwise, if `role` is specified, the result is the pipeline through the first protocol performing that role. This can optionally be qualified by `method`. If neither `role` nor `outtype` are specified, the first `n` elements are returned, as usual.

`tail(x, n = 6L, role, method = character(), intype)`: Like ordinary `tail`, takes a suffix of the pipeline. If `intype` is provided, this returns the pipeline starting at the last protocol with `intype` as its input type. Otherwise, if `role` is specified, the result is the pipeline starting at the last protocol performing that role. This can optionally be qualified by `method`. If neither `role` nor `outtype` are specified, the last `n` elements are returned, as usual.

`Pipeline` extends `list`, so, in general, it can be treated as one when it comes to subsetting, etc. For example, `[` method works for subsetting of `Pipeline` object.

Utilities

`findProtocols(object, role, method = character())`: Get the indices of the protocols in the pipeline with the specified `role` and `method`.

Author(s)

Michael Lawrence

Examples

```
setStage("average", intype = "numeric")
setProtocol("mean", fun = mean, parent = "average")
setProtocol("quantile", representation = list(probs = "numeric"),
            fun = quantile, parent = "average")
setProtocol("range", representation = list(low = "numeric", high = "numeric"),
            fun = function(x, low = 0, high = Inf) x[x >= low & x <= high],
            parent = setStage("trim", intype = "numeric"))

d <- c(1, 2, 4)
p <- Pipeline("trim", "average")
perform(p, d)

p <- Pipeline(Protocol("trim", low = 2), "average")
perform(p, d)

p <- Pipeline(Protocol("trim", low = 2),
              Protocol("average", "quantile", probs = 0.75),
              displayName = "Filter and Average")
perform(p, d)

## accessor
inType(p)
outType(p)
parameters(p)
protocol(p, "average")
protocol(p, "average", "quantile")
displayName(p)

## utils
findProtocols(p, "average")

## subsetting
# make a new example
setStage("DemoCastN2C", intype = "numeric", outtype = "character")
setProtocol("cast", fun = function(x){
               message("Convert from numeric to character")
               as.character(x)
            },
            parent = "DemoCastN2C")

setStage("DemoCastC2F", intype = "character", outtype = "factor")
setProtocol("cast", fun = function(x){
               message("Convert from character to factor")
               as.factor(x)
            },
            parent = "DemoCastC2F")

setStage("DemoCastF2L", intype = "factor", outtype = "list")
setProtocol("cast", fun = function(x){
               message("Convert from factor to list")
               as.list(x)
            },
            parent = "DemoCastF2L")

d <- 1:3
p <- Pipeline(Protocol("DemoCastN2C"),
              Protocol("DemoCastC2F"),
              Protocol("DemoCastF2L"))
p
perform(p, d)
# subsetting
# convert to a factor
p12 <- p[1:2]
p12
perform(p12, d)

#
p23 <- pipeline(p, intype = "character")
p23
perform(p23, as.character(d))

#
p12 <- head(p, 2)
p12
#or
head(p, outtype = "factor")
head(p, role = "DemoCastC2F")

tail(p, 2)
tail(p, intype = "character")
tail(p, intype = "factor")
tail(p, role = "DemoCastC2F")

#combination
p1 <- Pipeline(Protocol("DemoCastN2C"))
p2 <- Pipeline(Protocol("DemoCastC2F"))
p3 <- Pipeline(Protocol("DemoCastF2L"))
c(p1 ,p2)
p[2] <- p2

setClass("ExChar", contains = "character")

setStage("DemoCastC2FV2", intype = "ExChar", outtype = "factor")
setProtocol("cast", fun = function(x){
               as.factor(x)
            },
            parent = "DemoCastC2FV2")

p4 <- Pipeline(Protocol("DemoCastC2FV2"))

## Not run:
## doesn't work, input 'charcter' is super class of output 'ExChar'.
p[2] <- p4
```

```
## End(Not run)
p

## as a subclass, works.
setStage("DemoCastN2CV2", intype = "numeric", outtype = "ExChar")
setProtocol("cast", fun = function(x){
               new("ExChar", as.character(x))
            },
            parent = "DemoCastN2CV2")
p5 <- Pipeline(Protocol("DemoCastN2CV2"))
p[1] <- p5
p

## Not run:
## won't work, because the outtype doesn't match the intype.
c(p1, p3, p2)
p[c(1, 3)]
p[2] <- p3

## End(Not run)
```

`PipelineData-class` *PipelineData: Data with history*

Description

`PipelineData` is a virtual class representing a dataset with an attached pipeline that describes the series of steps that produced the object. The storage of the data is up to the implementation. The methods described here apply equally to `PipelineData` and any other object that has `pipeline` as a slot/attribute.

Methods

`pipeline(object, ancestry = TRUE, local = TRUE)`: Gets the pipeline that produced the object. If `ancestry` is `TRUE`, the returned pipeline includes the protocols that produced predecessors of a different type. If `local` is `TRUE`, the pipeline includes protocols after the last protocol that output an object of a different type, i.e., all local protocols have this type as both their input and output.

`explore(object)`: Produces an interactive, exploratory visualization of this data, in the context of the last applied protocol.

Author(s)

Michael Lawrence

Examples

```
## A non-PipelineData data example
setStage("average", intype = "numeric")
setProtocol("mean", fun = mean, parent = "average")
setProtocol("quantile", representation = list(probs = "numeric"),
            fun = quantile, parent = "average")
setProtocol("range", representation = list(low = "numeric", high = "numeric"),
            fun = function(x, low = 0, high = Inf) x[x >= low & x <= high],
            parent = setStage("trim", intype = "numeric"))

d <- c(1, 2, 4)
p <- Pipeline("trim", "average")
d2 <- perform(p, d)
attr(d2, 'pipeline')
pipeline(d2)
## Not run:
## this will give an error, no slot called pipelinem, just numeric value.
d2@pipeline

## End(Not run)

setClass("ProcessNumeric", contains = c("numeric", "PipelineData"))
d <- new("ProcessNumeric", c(1, 2, 4))
d@pipeline
setStage("average", intype = "ProcessNumeric")
setProtocol("mean", fun = function(x) new("ProcessNumeric", mean(x)), parent = "average")
setProtocol("quantile", representation = list(probs = "numeric"),
            fun = function(x) new("ProcessNumeric", quantile(x)), parent = "average")
setProtocol("range", representation = list(low = "numeric", high = "numeric"),
            fun = function(x, low = 0, high = Inf) new("ProcessNumeric",
                                              x[x >= low & x <= high]),
            parent = setStage("trim", intype = "ProcessNumeric"))

p <- Pipeline("trim", "average")
d2 <- perform(p, d)
attr(d2, 'pipeline')
pipeline(d2)
class(d2)
d2@pipeline
```

`Protocol-class` *Protocol: Concrete Step in a Pipeline*

Description

A `Protocol` object performs a `Stage` in a particular way, as part of a `Pipeline`. Most users will simply construct a `Protocol` and add it to a pipeline. To define a new type of `Protocol`, see `setProtocol`.

Constructors

`Protocol(role, method = defaultMethod(role), ...)`: Creates a protocol of the stage identified by `role` and method given by `method`. The `role` argument may be either a `Stage` object or a string naming the role. Parameters in `...` are passed to the initializer, i.e., they specify parameters of the protocol by name.

Accessors

`stage(object, where = topenv(parent.frame()))`: Return a `Stage` object that represents the role this protocol plays in a pipeline. Searches for the class definition of the stage in `where`.

`method(object, where = topenv(parent.frame()))`: Returns the method name of this protocol. This is derived from the class name of the protocol by removing the stage name. The environment `where` should contain the definition of the stage class.

`displayName(object)`: Gets the name for displaying this protocol in a user interface.

`inType(object)`: Gets the class of data that this protocol accepts as input.

`outType(object)`: Gets the class of data that this protocol yields as output.

`parameters(object)`: Gets the list of parameters, i.e., the slots of the object, that control the execution of the protocol.

`pipeline(object)`: Some protocols delegate to a secondary pipeline, i.e., they have a slot named "pipeline". This function retrieves that, or returns `NULL` if the protocol does not delegate to a pipeline.

Author(s)

Michael Lawrence

See Also

`setProtocol` for defining new types of protocols

Examples

```
setStage("average", intype = "numeric")
setProtocol("mean", fun = mean, parent = "average")
setProtocol("quantile", representation = list(probs = "numeric"),
            fun = quantile, parent = "average")

proto_avg_mean <- Protocol("average")
proto_avg_mean <- Protocol("average", "mean")
proto_avg_quantile <- Protocol("average", "quantile")
proto_avg_quantile_075 <- Protocol("average", "quantile", probs = 0.75)

proto <- proto_avg_quantile_075
proto
stage(proto)
inType(proto)
parameters(proto)
```

`setProtocol` *Define a Protocol Type*

Description

This function defines new derivatives of the `Protocol` class. It is a wrapper around `setClass` and thus has a very similar interface.

Usage

```
setProtocol(method, dispname = method, representation = list(), fun,
            parent, prototype = list(), validity = NULL,
            where = topenv(parent.frame()))
```

Arguments

`method`	The name of the method performed by protocols of this type
`dispname`	The display name for protocols of this type
`representation`	A list declaring the names and types of the parameters, implemented as slots in the S4 class
`fun`	The function implementing the protocol. If omitted, this protocol type will be virtual. This function will be passed the input data, any parameters named in its formals, and any arguments passed to `perform`. Default values for its arguments override values in `prototype`. Use `callNextProtocol` to chain up to the implementation of a parent protocol.
`parent`	The single parent/super class for this protocol class. Usually, this is the role, i.e., the name of the `Stage` for this protocol type. Also could be the name of a class inheriting from `Protocol`, or the concantenation of the role and method names.
`prototype`	As in `setClass`, the list indicating initial values for the parameters/slots. Usually not necessary, because it is derived from the formals of `fun`.
`validity`	The function for checking the validity of an object, see `setClass`.
`where`	The environment in which this protocol class is defined.

Details

Every type of protocol in a pipeline is implemented as an S4 class, ultimately derived from `Protocol`. The parameters controlling the execution of the protocol are represented by slots in that S4 class.

Through S4 inheritance, each protocol is associated with a `Stage`, which represents the role a protocol plays in the pipeline. For example, a protocol might have an “average” stage, with two protocols: “mean” and “median”. Here, “average” would be the role name and would have an associated `Stage` derivative. Meanwhile, “mean” and “median” are `method` names and would each have a corresponding `Protocol` derivative. Protocols that have the same stage all derive from a common, virtual `Protocol` derivative corresponding to that stage. In our example, we would have two protocol classes: `ProtoAverageMean` and `ProtoAverageMedian`. Both would inherit from `ProtoAverage`, which in turn inherits from `Protocol`.

Another side effect of this function is that a generic is defined, named of the form `role.Method`, that performs this protocol, given the data and additional arguments. There is a method for the `inType` of the stage. Thus, in our example, we would have generics `average.mean` and `average.median`.

Value

The name of the class

Author(s)

Michael Lawrence

See Also

`Protocol` for constructing protocol objects, `setStage` for defining `Stage` classes.

Examples

```
setStage("average")
setProtocol("mean", fun = mean, parent = "average")
setProtocol("median", fun = median, parent = "average")
d <- c(1, 2, 4)
average(d)
average(d, "median")
average.median(d)
```

`setStage` *Define a Stage Class*

Description

This function defines new derivatives of the `Stage` class. It is a wrapper around `setClass` and thus has a similar interface.

Usage

```
setStage(name, dispname = name, intype = "ANY", outtype = intype,
         where = topenv(parent.frame()))
```

Arguments

`name`	The name of the stage, i.e., the role string
`dispname`	The name for display in a user interface
`intype`	The class of the data that protocols of this stage accept as input
`outtype`	The class of the data that protocols of this stage accept as output
`where`	The environment in which to define the stage class

Details

Calling `setStage` defines two classes:

- A derivative of `Stage` that represents this stage
- A derivative of `Protocol` from which all protocols that implement this stage derive

For example, if we had a stage named "average", calling `setStage` would create the classes `StageAverage` and `ProtoAverage`.

The function also defines a generic, named by the `name` argument, that performs a protocol of this stage. There is a method that takes an object of type `intype` as first argument and a method name as its second. Additional arguments are passed to the `perform` method of the protocol. In our prior example, there would be a generic called `average` and method `average,numeric` if `intype` was given as "numeric".

It also defines a generic of the form `nameProto` that serves as an accessor for protocols of this stage. A method is defined for `Pipeline` and `outtype`, so that one could retrieve our "average" protocol with `averageProto(pipeline)` or `averageProto(result)`. Similarly, a replacement generic and methods are defined.

Value

The name of the role

Author(s)

Michael Lawrence

See Also

The `Stage` class; `setProtocol` for defining a new type of protocol

Examples

```
## simplest definition
setStage("average")
## add a display name and specialize to numeric input
setStage("average", "Average Vector", intype = "numeric")
setProtocol("mean", fun = mean, parent = "average")
setProtocol("quantile", representation = list(probs = "numeric"),
            fun = quantile, parent = "average")
setProtocol("range", representation = list(low = "numeric", high = "numeric"),
            fun = function(x, low = 0, high = Inf) x[x >= low & x <= high],
            parent = setStage("trim", intype = "numeric"))

## Class Stage derivative
showClass("StageAverage")
## Class Protocol derivative
showClass("ProtoAverage")

## generic defined
showMethods("average")
```

```
# try this generic method
d <- c(1, 2, 4)
average(d, "mean")

## create a pipeline
p <- Pipeline("trim", "average")
res <- perform(p, d)
res
## generic *Proto
averageProto(p)
averageProto(res)
```

Stage-class *Stage: Abstract Step in a Pipeline*

Description

A `Stage` object represents a role to be played by protocols in a pipeline. In other words, a stage is an abstract step that transforms one data type into another. The transformation may be implemented in a number of ways, each corresponding to a protocol. Users normally do not have to interact with this object. Developers can define new types of stages with `setStage`.

Constructors

`Stage(role)`: Creates a stage object given the `role` name.

Accessors

`role(object)`: Gets the name of the role represented by this stage.

`displayName(object)`: Gets the name for displaying this stage in a user interface.

`inType(object)`: Gets the class of data that protocols of this stage accept as input.

`outType(object)`: Gets the class of data that protocols of this stage yield as output.

`defaultMethod(object)`: Gets the name of the default method associated with the role of this stage. If not explicitly set, this becomes the first protocol registered for the stage.

`defaultMethod(object) <- value`: Sets the name of the default method associated with the role of this stage.

`methodNames(object, where = topenv(parent.frame()))`: Gets the names of the methods for this stage, looking in `where` for the protocol classes.

Author(s)

Michael Lawrence

See Also

`setStage` for defining new types of stages

Examples

```
setStage("average", "Average Numbers", intype = "numeric")
setProtocol("mean", fun = mean, parent = "average")
setProtocol("median", fun = median, parent = "average")

stage <- Stage("average")
stage

defaultMethod(stage)
defaultMethod(stage) <- "median"
defaultMethod(stage)
```

Index