Taxonomy of Roxygen Directive Kinds

Melina Vidoni (Australian National University), Zadia Codabux (University of Saskatchewan)

This taxonomy is presented in Figure 1 and is colour-coded. In the Figure, green labels represent the segments studied in this manuscript, while the blue ones are the groups of directives (those shared between multiple segment types are indicated with a blue share icon). Grey squares represent possible directives (if they are exclusive, they are indicated with a |, or with a red lock if restricted to which segment uses it).

Based on previous definitions Monperrus et al. [1], a **directive** is *a natural-language statement that makes developers aware of constraints and guidelines related to the correct and optimal use of an R Package.* In contrast, a **directive kind** is *a set of directives that share the same kind of constraints or guidelines.*

Each of the *directive kinds* is presented using the same pattern proposed by Monperrus et al. [1]:

- **Name:** A short name to identify the directive kind that summarises what it represents.
- **Definition:** The explanation of the directive. Usually, the first paragraph after the title, with no additional sub-title.
- **Discussion:** The rationale behind the existence of the directive kind, relevant observations, and any concerns related exclusively to R.
- **Example:** Derived from the dataset, that are a prime example for this type. If omitted, they are given in the sections corresponding to good practices or anti-patterns.
- **Good Practices:** Optional. Represents good practices that clarify information about the directive.
- **Anti-Patterns:** Optional. These are trends that are usually ineffective and risk being highly counterproductive when using a directive kind [2].

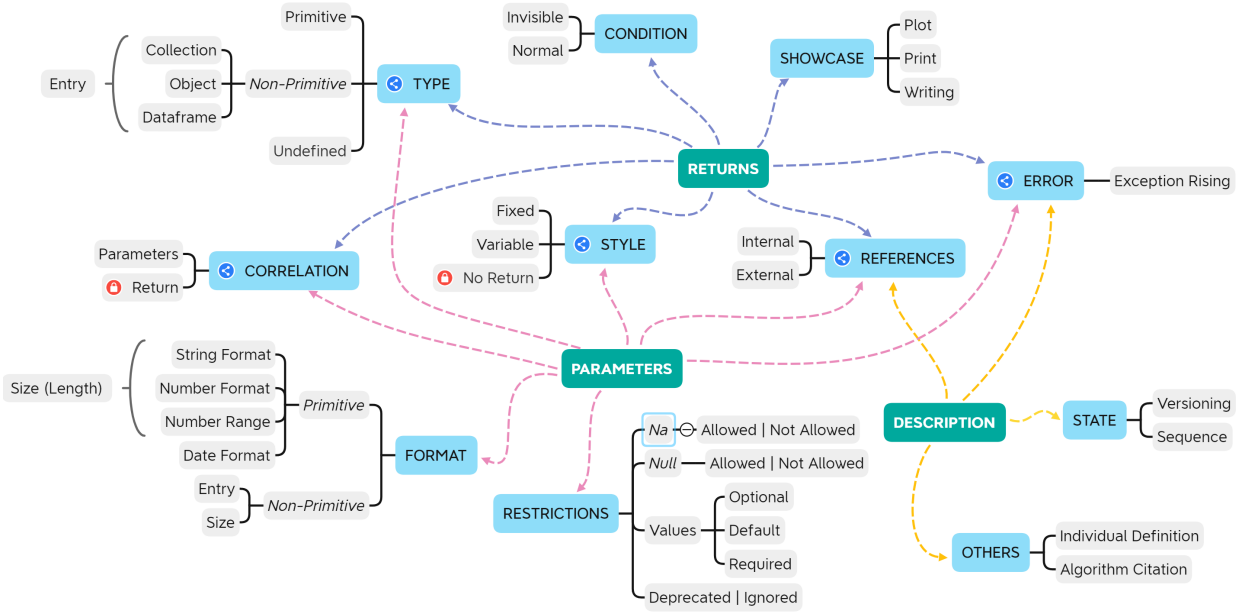

Figure 1: Complete Taxonomy of Roxygen Directives

1 Return-Exclusive Directives

These are found exclusively on the `@return` segments. They express constraints and guidelines when documenting a function's return–the term 'function' is preferred as this tag can be used for regular functions or R's OO methods alike. These relationships between directives and *directive kinds* are summarised in Figure 2, which is colour-coded. Some directives can appear in combinations (highlighted as +combo); while other directives determined which could be used in another *directive kind* (e.g., Style and Condition) and the sequence is indicated with dashed arrows. Some **conditionally shared** appeared in combination with other

directives; this is highlighted as **>may be affected by**, and a dotted arrow to the related directive.

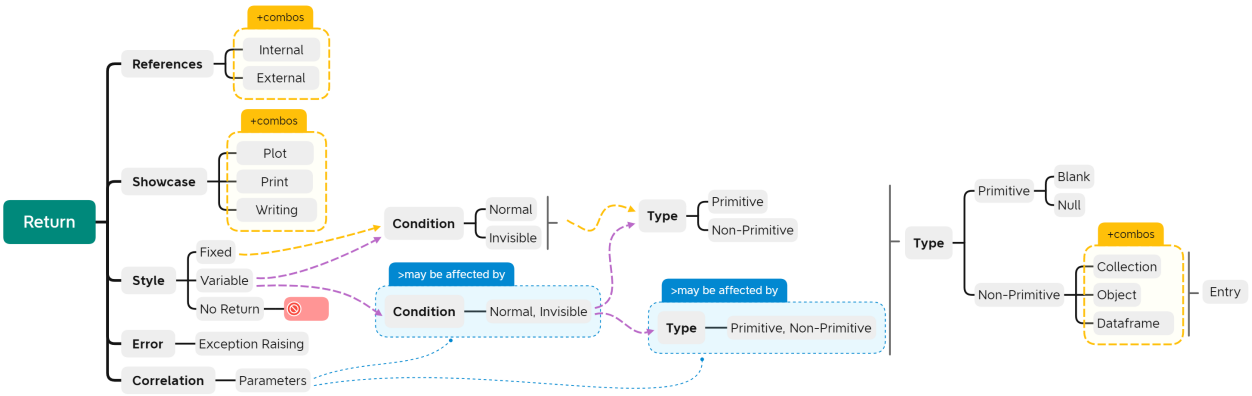

Figure 2: Return directives relationships and limitations.

1.1 Condition

These express how a return is being rebounded and are exclusive to the returns.

1.1.1 Invisible

A Condition directive. It states that a return is invisible. They are sometimes described as 'silent return'.

Discussion. The developer must use the `invisible()` expression [3], which is temporarily prevented from being printed out [4]. An invisible return does not stop the execution of a function unless combined with a regular return. As R scripts and markdown reports rely on printing values, making this explicit is critical. It is possible to be combined with Style>Variable, to have an invisible return happening only under specific conditions. It can also be related to a parameters Correlation>Return.

Examples. As a clarification: `@return the name of the temporary table created (invisibly)`. As a clear explanation: `@return invisibly returns the given karyoplot object`. With fixed returns, such as `@return TRUE (invisibly)`.

Anti-Patterns. Not specifying that all returns are invisible or not; e.g.,`@return invisibly returns the order of rows, if clustfun is provided and/or order=TRUE`, since there is no information about what is returned in other cases, and if it is invisible.

1.1.2 Normal

A Condition directive. It states that a return is not invisible, regardless of how it was returned.

Discussion. It happens when a developer uses use `return(...)` to stop the execution and rebound the value passed there, or lets the function finish and return the last in-scope assigned variable. This is not related to the returned type, and can be combined with Type to explain that. It is possible to be combined with Style>Variable, to have a normal return happening only under specific conditions, or even change the type being rebounded. It can also be related to a parameters Correlation>Return.

Examples. A clarification on the conditions for a variable return (primitive and non-primitive, but always visible): `@return A POSIXct object if successful, otherwise failure`. An example of fixed primitive return: `@return A string giving the complete mime type, with all parameters stripped off`.

Anti-Patterns. Leaving the return blank where there are returned values. Not specifying all types of return, or providing incomplete information: `@return A list of graph attributes, or a single graph attribute`; another example, but Style>Fixed, is `@return A new graph object` or `@return a tibble`.

1.2 Showcase

In R, it is often common to provide alternative returns that are not traditionally rebounded (using the Condition directives). These may be complementary or completely replace the above (e.g., a function with Style>No Return can still Showcase).

1.2.1 Plot and Print

A SHOWCASE directive. It states that a specific part of the return is written in the console (either printed or logged) or plotted into the inspector.

Discussion. Developers can print or log by using expressions such as `print()`, `message()` or `error()` [3]. Though most plots can be returned as objects (e.g. 'ggplot2'), if not assigned, they are displayed immediately in the inspector [4]. Since many functions rely on this, using this directive is essential to clarify the default behaviour.

Examples. To indicate that something is plotted: `@return None. Function produces a plot`. To indicate that the output is printed: `@return None. Results are printed`. Combined with directives of the *Style* group: `@return Prints the PharmacoSet object to the output stream, and returns invisible null`.

Good Practices. It must be used in combination with STYLE and TYPE directives to clarify any other returns or lack thereof (style) and the type that is being used. If the function has only a showcase in combination with a STYLE>NO RETURN directive, then the *Type* group can be disregarded. Must clarify if the showcase is printed and returned simultaneously.

Anti-patterns. In R, plot objects will be printed if they are called; thus, the directive should be clear regarding automatically plotting and returning the object. For instance, `@return ggplot object that if called, will print` is stating a property of a plot object, instead of what the function returns and showcases.

1.2.2 Writing

A SHOWCASE directive, where part (or all) of the output is saved as a file at a specific path.

Discussion. Writing an outcome at a specific path is often helpful to avoid losing a processed data set due to an error on the IDE (Integrated Development Environment).

Examples. Combining writing a file and returning an output (STYLE and TYPE): `@return Returns a list of metrics derived from the simulated full waveform. A text file (txt) containing the metrics will be saved in the output folder (outRoot)`.

Good Practices. Explain if any directories are produced or must already exist; e.g., `@return For type='sparse', a directory is produced at 'path' [...]`. If the writing path is a default one, it should also be clarified here. If a parameter is the path, the full explanation should be given in the parameter segment. Since this can occur alongside a return, a WRITING must be used with STYLE and TYPE directives. `@return An invisible data list, and a file is written to the disk if an entry other than the default of NULL is provided for outfile`, shows that, if writing the file depends on parameters, this information must be given.

Anti-Patterns. Not clarifying what path is used to save the file. Unclear wording about the file being written, returned as an object, or both: `@return A modified SS .dat file, and that file returned invisibly (for testing) as a vector of character lines`.

2 Parameters-Exclusive Directives

These are directives found on the `@param [name]` segments. They express constraints and guidelines when documenting a specific argument for a function; the term 'function' is preferred as this tag can be used for regular functions or R's OO methods. These are summarised in Figure 3, which is colour-coded. Some directives can appear in combinations (highlighted as +combo); while other directives determined which could be used in another *directive kind* (e.g., STYLE and CONDITION) and the sequence is indicated with dashed arrows. Some **conditionally shared** appeared in combination with other directives; this is highlighted as **>may be affected by**, and an dotted arrow to the related directive. Additionally, muatually **exclusive** directives are boxed in.

2.1 Restrictions

These refer to multiple restrictions enforced in a parameter, either by documentation or through a function's internal logic. While some of these can be added into the signature of a function (i.e., a DEFAULT value), most are only enforced through internal behaviour that must be clarified for the sake of correct usage.

2.1.1 Null Allowed, Null Not Allowed

A RESTRICTION directive, specifies if a parameter is allowed (or not) to be `null`. It also explains the specific semantics of the `null` value for the respective parameter [1], and any impact it may have on the function's

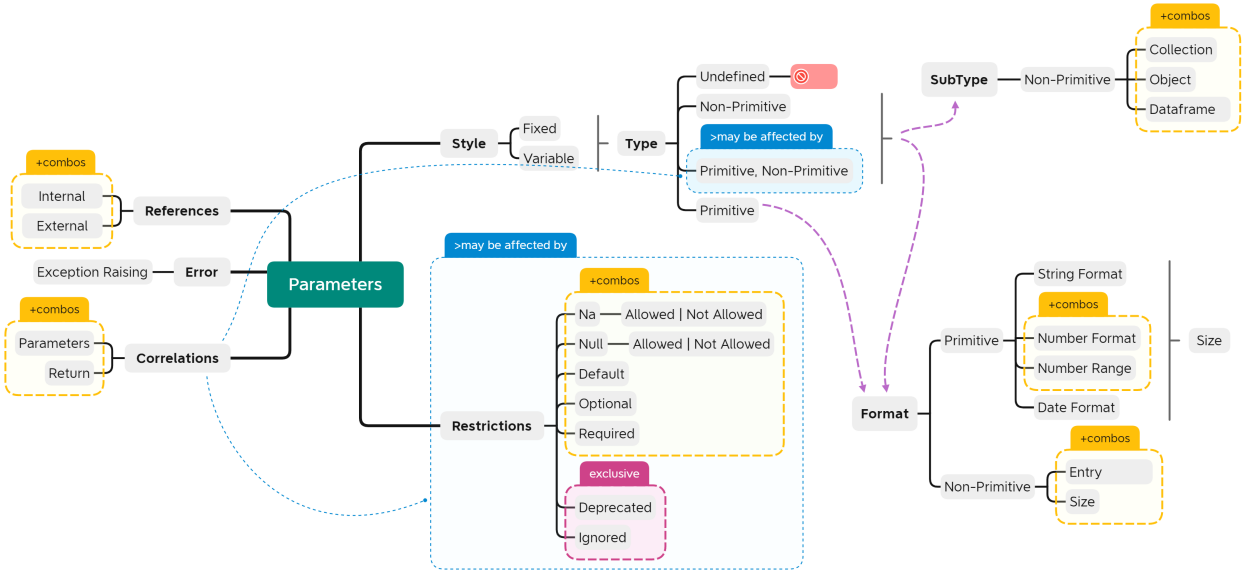

Figure 3: Parameters directives relationships and limitations.

behaviour and return. It is derived from the work of Monperrus et al. [1].

Discussion. `null` represents a value that does not exist, such as an empty object, indicating an undefined value [3]. This directive states if a `null` value is allowed (and its specific semantics and impact), or if it is not. It combines with TYPE (it can be a TYPE>PRIMITIVE that allows a `null` value, or a TYPE>NON-PRIMITIVE that allows `null` values on its entries). A parameter can allow `NA` and `null` at the same time.

It is also possible to combine this with other restrictions, such as CORRELATION>PARAMETERS (e.g., can or cannot be `null` when another argument takes specific values or is used), and also with CORRELATION>RETURNS (i.e., a `null` can affect a result). In some cases, it may be related to ERROR>EXCEPTION RAISING as if a `null` value is received but not allowed, the function may throw an error. All these cases should be properly documented and explained, or they become anti-patterns.

Example. Stating that `null` is allowed and its effect, and used in combination with NA RESTRICTIONS directive: `@param title The title of the plot. null eliminates title. NA uses the title attribute of the Network object.`. The NULL ALLOWED can be used in combination with a DEFAULT (e.g., the default value is `null`), such as: `@param container [...] Defaults to 'NULL'.` or by parsing its value: `@param title.cex Character expansion factor for the title. NULL and NA are equivalent to 1.0`.

Good Practices & Anti-Patterns. Clarify the effect of a `null` argument value: `@param x ff object where data will be appended to. If x==NULL a new ff object will be created`. An anti-pattern is allowing `null` without explaining the semantics nor effects: `@param vp a grid viewport object (or NULL)`.

2.1.2 NA Allowed, NA Not Allowed

A RESTRICTION directive. It specifies if a method parameter is allowed (or not) to be NA. It also explains the specific semantics of the NA value for the respective parameter and its impact on the function's behaviour or return. This was extended from the NULL ALLOWED, NOT ALLOWED directive from the work of Monperrus et al. [1], given that it acts in the same way but only with R's unique value `NA`.

Discussion. In R, `NA` is a reserved word, with a constant that indicates missing values in any type[1]. It is not the same as a `null` value because `NA` can be accessed and managed [3]. This directive clarifies if the parameter can receive a `NA` value or if it is forbidden. If combined with TYPE>NON-PRIMITIVE, it should detail if some elements can be NA and what happens to the whole element; it may also refer to inner `NA` (e.g., a matrix that accepts missing values). Hence, developers also refer to `NA` as 'missing' or 'empty values' (according to how it is presented in books and samples).

Like with the NULL ALLOWED, NOT ALLOWED directive, it can be combined with both CORRELATION, with RESTRICTION>DEFAULT or even ERROR>EXCEPTION RAISING. All these cases should be properly documented and explained, or they become anti-patterns.

Example. If the text is clear enough, the reserved word does not need to be included: `@param Data A`

[1]https://www.rdocumentation.org/packages/base/versions/3.6.2/topics/NA

`numeric matrix or data frame (which may contain missing values)`.

Good Practices. Clearly stating what happens in the function behaviour when a `NA` value is passed, such as in :`@param dates [...] If NA, this means use the lower/upper limit as appropriate`).

Anti-Patterns. Stating that a `NA` is allowed, but not explaining the semantics of passing it, nor its effect on the results. When dealing with NON-PRIMITIVE directives, not clarifying if the object as a whole or its elements can be `NA`.

2.1.3 Default

A RESTRICTION directive, stating that an argument has a default value to be used if nothing is passed there. It explains the semantics of the 'default' and its impact on the function's behaviour and return. It may appear alongside RESTRICTIONS>OPTIONAL or complement it, but that is not always the case.

Discussion. The default value is stated in the signature of a function, as `argumentName = 'defaultValue'`, and means that if the argument is not explicitly passed upon invocation, the functions' behaviour will revert to using whatever was set as the default value [3]. Default values can be of any type, `NA` or `null` (thus being combined with the corresponding directives). In those cases, the practices and anti-patterns of NA and NULL directive restrictions apply. A default value may not be explicit in the signature but calculated in the function if the argument was empty or null; in this case, it will not appear in the documentation and must be documented.

Example. When specifying a value: `@param saTemp1 Final temperature for SA (default = 0.01)`. When the default value is `null`: `@param grouping.var The grouping variables. Default NULL generates one word list for all text`.

Good Practices & Anti-Patterns. If the default is internally calculated and cannot be explicitly written in the signature, it should be explained. An anti-pattern is not including the semantics and impact of a default: `@param legend Plot legend, default = TRUE`. Detailing the behaviour of the default value will be considered good practise; for example `@param is_EM Is this an estimation model? Defaults to NULL, which will look for the letters "em" (lower or uppercase) to decide if this is an estimation model or operating model`.

2.1.4 Optional

This is a RESTRICTION directive that often appears with RESTRICTIONS>DEFAULT but may also appear on its own. It refers to parameters that may be omitted for many reasons.

Discussion. In R arguments are optional, and a function does not need to be invoked with all of them as long as the few used arguments are explicitly named on the invocation [3]. Ideally, when an argument is OPTIONAL, it will also have a DEFAULT value that would be used if nothing is received. However, it is possible to have optional arguments not used if nothing is passed. Moreover, given that there are CORRELATION>PARAMETERS (e.g., using a parameter affects how another argument of the same function is used), some arguments can become optional if another is passed. Likewise, they can become REQUIRED, making it possible for an argument to be labelled as both OPTIONAL and REQUIRED at the same time.

Examples. A case of an optional argument without default that is ignored if not passed: `@param report Optional name of an html file for generating reports`.

Good Practices. If the argument is optional under some conditions but required in others, the conditions (or related parameters, if existent) must be explicitly stated and clarified. For example: `@param longitude,latitude optional signed numbers indicating the longitude in degrees East and latitude in degrees North. These values are used if 'type="sippican"', but ignored if 'type="noaa1"', because those files contain location information`.

Anti-Patterns. If the argument is optional but has a default value, all the anti-patterns of DEFAULT also apply here. If multiple parameters can be passed into a single argument, not providing detailed information about them is considered an anti-pattern; e.g., `@param Filters Optional parameters that let you set criteria the text must meet to be included in your response`. Not explaining what happens if the argument is not passed is an anti-pattern, e.g., `@param Clim Optional limit for conductivity axis`; in this case, the lack of effect or the argument being ignored should be stated.

When the OPTIONAL is applied to the ellipsis (...) argument, providing an REFERENCE>INTERNAL without properly linking it, is an anti-pattern; e.g., `@param ... Optional arguments. See Details`. In that case, all anti-patterns of this additional directive also apply here.

2.1.5 Required

This is a RESTRICTION directive that indicates that an argument is mandatory and must always have a value. However, it can be conditionally mandatory, enforced only under some conditions.

Discussion. This is the opposite of Restrictions>Optional as this argument must always receive a value. However, it can appear alongside it in situations where there is a Correlation>Parameters, as this parameter is mandatory or not depending on what is being passed. In many cases, it was found alongside Restriction>NA—Null not being allowed, given the function parsed `NA` or `null` as mandatory. Likewise, it can appear with Error>Exception Raising as the function may halt, throw or print an error if a mandatory parameter is missing.

Example. The mandatory condition should be explicit; e.g., `@param Identifier [required] The identifier for the resource server.`

Good Practices. If this is conditionally required, the condition should be explicit: `@param f A function to use for model fitting. Only required for GLM models at the moment`. If there are Correlation>Parameters that also affect the Format, this should be made explicit: `@param PrivateKey [required] The private key that matches the public key in the certificate`.

Anti-Pattern. Stating a requirement without explaining the Format directives: `@param DomainId [required] The domain ID`. Not explaining what happens to the functions' invocation when a Restriction>Required parameter is not passed or erroneous, e.g., `@param name [required] The name of the filter to create`. Likewise, if `NA` or `null` are considered as empty values, not explaining such interpretation in the documentation of the parameter is an anti-pattern. Finally, just stating that the argument is required without providing information on what should be passed, such as `@param GatewayARN [required]`.

2.1.6 Deprecated or Ignored

These are two Restriction directives that often appear together but are not the same. However, both point to arguments that are no longer used.

Discussion & Examples. Deprecated parameters are arguments that were used before but are kept for backwards compatibility or as a warning that they will soon be removed. In other cases, the arguments were replaced by new ones and disclosed in the description. In many cases, we found that deprecated arguments had the description deleted and only kept the status; for example, `@param method Deprecated`.

Ignored arguments are not necessarily deprecated but provide no information about what should be passed. In a few cases, this directive was used to highlight an argument to be implemented in the future while describing what it would use: `@param parShrinkMN a list for squeezeVar(). (NOT IMPLEMENTED)`; determining if this is an anti-pattern or not requires more occurrences and deeper analysis.

Good Practices & Anti-Patterns. If Deprecated, explaining whether this will be removed in the future or is kept for backwards compatibility should be clarified, e.g., `@param ... Ignored, included for S3 compatibility`; otherwise, it becomes an anti-pattern. If replaced by a new argument, the situation should be noted: `@param document **Deprecated** Use ‘devel‘ instead`.

Most cases of Ignored were ellipsis arguments (namely, `...`) explicitly ignored without providing an explanation, such as `@param ... other arguments (ignored)`; this is an anti-pattern as it clogs the signature and documentation, and provides no usability.

2.2 Format

These directives prescribe formats of particular arguments. As a result, they are exclusive to the parameters.

In general, they can appear alongside Type directives (which is why they share the first level). They may be combined with Style and have different requirements if they are Style>Variable, which should be explicit and detailed not to be an anti-pattern. Overall, they can also be combined with References for format guidelines (e.g., redirecting to an external website where the formats are listed, or to a shared document). Finally, they can also be combined with Correlation>Parameters (in case the format varies depending on when it is used), and Error>Exception Raising in case the format is not respected.

Combined anti-patterns. Given the number of combinations, all the good practices and anti-patterns of the associated or related directives will also apply to the Format directives. Additionally, updated or removed formats not reflected in the documentation, incomplete description of the constraint (e.g. specific values without describing their effects), or typos when listing options (thus rendering them incorrect).

2.2.1 String Format

Prescribes the format of a string argument and was derived from the work of Monperrus et al. [1].

Discussion & Examples. It constraints: lower/upper character, specific values to be accepted, accepted format (example: `@param URL Character. URL.`). Specific characters need to be escaped, such as `@param x Rd string. Backslashes must be double-escaped.`. Requiring some character types to be included is still part of this directive (e.g., a password with at least one special character). However, length

restrictions (i.e., no more or no less than a number) is considered Size (Length); e.g., `@param Description A description of the device. Length Constraints: Maximum length of 256 characters`.

2.2.2 Number Format

Prescribes the format of a number, including the type.

Discussion & Examples. In R, numbers can be integers, floating-point, precision, calculated, or even part of a factor. This directive constrains the type itself, e.g., `@param limit A integer. A limit of data in request`. It can also be used to prescribe specific values to be passed, such as `@param show Show labels, 1 or 0`. It may limit how many decimals a number may have, how the value is calculated or obtained, and it may refer to external documents; e.g., `@param number_format Format for numbering. See [number_format()] for details`.

Note that length or ranges limitations are part of the Format>Number Range directive. Prescriptions of maximum digits can also be interpreted as Format>Size(Length), such as: `@param domainOwner The 12-digit account number of the AWS account that owns the domain`.

2.2.3 Number Range

Used to delimit ranges in numbers [1], and it is often combined either with Format>Number Format or Format>Size(Length).

Discussion & Examples. It can prescribe both sizes of the range, with minimum and maximum values; e.g., `@param level A numerical value between 0 and 1 giving the confidence level`. However, it can also be used to specify only one of the ranges (e.g., by stating it must be positive); for example: `@param GlobalNetworkIds The IDs of one or more global networks. The maximum is 10`. It can also apply to individual elements of a collection of numbers, such as `@param probs numeric vector of probabilities with values in [0,1]`.

Sometimes, the rage was not written as an explicit range but embedded in the explanation of the calculation of the values; for example `@param maxJobDurationInSeconds [required] The maximum simulation job duration in seconds (up to 14 days or 1,209,600 seconds).`

2.2.4 Date Format

R dates have multiple formats, and different functions require unique structures [4]; in many cases, these formats depend on the packages that were used to parse and handle dates. This Format directive states a particular structure for a date. This directive covers date alone, time alone, and date-time formats.

Discussion & Examples. Stating the format by referring to the date object that should be passed, e.g., `@param DeferMaintenanceStartTime A timestamp indicating the start time for the deferred maintenance window`. Strings that only contain dates are considered under the Date Format directive, and not the String Format directive; for example: `@param date Date for which to get schedule (YYYY-MM-DD)`.

Good-Practices. In the case of complex structures, providing an example of a good value is a good practice. For example: `@param endTime The end time of the time period for the returned time series values. This is specified using the ISO 8601 format. For example, 2020-06-01T13:15:02.001Z represents 1 millisecond past June 1, 2020 1:15:02 PM UTC.`

Anti-Pattern. When referring to objects that represent dates, not providing a reference to a said object is an anti-pattern. Moreover, dates sometimes require timezones to be properly handled; not stating whether a timezone is relevant or not, and if they are converted or manipulated as-is, is also an anti-pattern, as it may provide incorrect results. If incorrect formats are forcefully parsed, this should be stated alongside its effects to disclose them and allow debugging if needed.

2.2.5 Size or Size (Length)

This directive takes different names depending on its association to Type>Primitive (in which it refers to the length, and was explained in the other formats), or Type>Non-Primitive.

Description & Examples. When appearing with Type>Non-Primitive, it refers to limits to collections, bytes, matrix dimensions (without naming the columns), and similar. It can also be combined with Format>Non-Primitive>Entry.

In the case of a Type>Primitive, `@param season: A 4-digit year associated with a given NFL season`. When limiting by size: `@param Text [...] Each string must contain fewer than 20,000 bytes of characters`. When limiting the size of a collection: `@param jobDefinitions A list of up to 100 job definition names or full Amazon Resource Name (ARN) entries`. When stating

the size of a matrix: `@param network matrix n1*n2`.

2.2.6 Entry

Similar to TYPE>NON-PRIMITIVE>ENTRY, it refers to the particular format of non-primitive parameters (e.g., vector variable names, dataframe columns, object attributes).

3 Description-Exclusive Directives

These were found exclusively on the `@description` segments, and are summarised in Figure 4, which is colour-coded. Some directives can appear in combinations (highlighted as +combo). Although the tag `@description` is optional, this part of the taxonomy only covered those segments properly tagged; as a result, these directives only cover part of what can be discussed in the description of a function when using Roxygen. Note that descriptions can be quite long. As a result, the examples will be excerpts, using a "[...]" to indicate when a text fragment has been extracted.

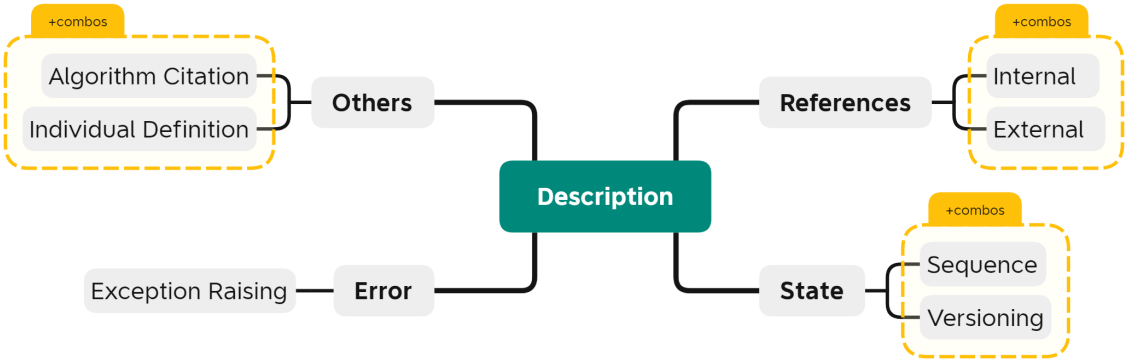

Figure 4: Description directives relationships and limitations.

3.1 State

The name for this group of directives was chosen since it encompasses both the version (e.g., a version in a 'stable' state) and the sequence of use (e.g., a function used in the 'preliminary' state of the sequence).

3.1.1 Sequence

It specifies the order of method calls and derives from the work of Monperrus et al. [1].

Discussion. The sequence does not need to be mandatory, and methods can be optionally related. It can indicate which internal method will be used. Using an INTERNAL REFERENCE directive is implied, and it is affected by those good practices and anti-patterns. It can be used with other directives.

Examples. A case of mandatory sequence, with an unliked internal reference: `@description Workhorse for posterior adaptive grid approximation. Called from cause_grid_adapt`. Indicating an optional sequence: `@description Creates a learning curve object, which can be plotted using the plotLearningCurve() function`.

Anti-Patterns. Unclear sequence: `@description A helper function prepares a working directory for running an analysis with CAUSE`.

3.1.2 Versioning

It indicates the lifecycle state of a function. It does not require an explanation of changes or bugs but should provide replacement functions.

Discussion. A function may be deprecated or 'experimental'. This directive does not require additional information about the state but contributes to the dependencies' stability (e.g., moving away from deprecated or unstable functions).

Examples. An experimental function may not work properly: `@description \lifecycle{experimental} Shows the variable names that are in common between two or more tibbles.`

Good Practices. Roxygen provides several tags for this, such as `r lifecycle::badge("stateName")`

or the latex-like command `\lifecycle{state}`. Since this is automatically parsed when building the documentation available in the IDEs, its use is encouraged [3]. If a function is deprecated a new one is available, the directive should include a link.

Anti-Patterns. Not mentioning nor linking to alternative functions if deprecated: `@description DEPRECATED`.

3.2 Others

These are additional directives that were detected in the analysed description fragments.

3.2.1 Algorithm Citation

It specifies an algorithm implemented in the function. It can mention the name (for a well-known and established algorithm) or provide a citation.

Discussion. Package citations and scientometrics of packages are important [5, 6], and given R's scientific use, many packages implement existing algorithms [7, 8]; e.g., many academic works compare different implementations in R of the same algorithms [9–11]. This directive indicates the algorithm used and how it works. For instance, `@description Calculates the McCune & Keon (2002) Heat Load Index`.

Good Practices. Using an academic citation with linked DOI or referencing to a 'citation' internal documentation page, where all relevant bibliographies are linked.

Anti-Patterns. Using academic citations and not providing a DOI hinders understandability and reproducibility. Mentioning an algorithm by name without providing a link or citation is taxing for developers working on disciplines with many algorithms available. Moreover, finding the specific manuscript may not be possible when papers are cited only by authors' names and publication date.

3.2.2 Individual Definition

Clarifies individual behaviour of every function in a family or group. It is only applicable to shared or grouped documents.

Discussion. Roxygen allows summarising the documentation of a family or group of functions that have similar behaviour, arguments or return, in order to simplify its readability and browsing [3]. A proper example is the official documentation of the base family of `lapply`[2].

Good Practices. Detailing each function while summarising commonalities. Using formatted syntax is encouraged. For getters and setters, a summarised explanation is acceptable if there is no data manipulation; e.g., `@description 'maxFeatures','maxFeatures<-': getter and setter for the'maxFeatures' slot of the object.`

Anti-Patterns. Many functions are listed, but the individual description replicates other functions without new information. When a group is composed of several functions, only some of them are explained.

4 Shared Directives

These directives were found in two or three segments and are summarised in Figure 5. This figure does not explain the relationships between them, as they were mentioned in the Sections above.

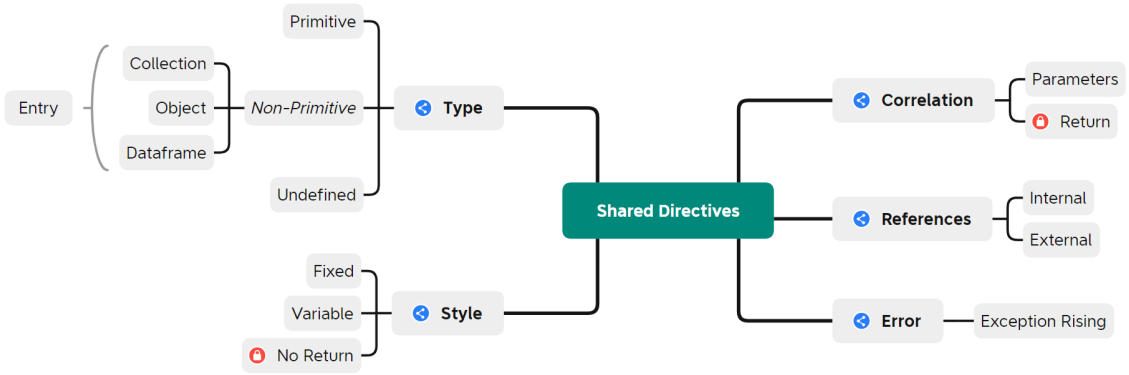

Figure 5: Shared directives detected in the taxonomy.

[2]https://tinyurl.com/lapplydoc

4.1 References

These directives refer to the cases where there are pointers to additional resources. They can be either Internal or External, and wee detected among the parameters, returns and descriptions.

4.1.1 External

A Reference directive, it points to an external source (not generated by the current documentation) to clarify constraints on an argument. This is a dual directive to Internal reference directive.

Discussion. Roxygen allows linking to other packages, which is essential given the nature of dependencies networks in R, how they affect package growth [5], and the widespread use of functions clones [12]. It can be used to indicate an external source (i.e. using `\url{...}` or markdown syntax `[]()`) that clarifies which arguments can be accepted; this is useful for packages connecting to external APIs.

Examples. To indicate a list of accepted argument values, determined by an API: `@param ... Additional named values that are interpreted as Quandl API parameters. Please see (working URL) for a full list of parameters`. To indicate sources of data, such as `[...] The data set resembles the [chondro](https://...) data set but is entirely synthetic`.

Anti-Patterns. Due to Roxygen's alternative syntaxes available to embed a working link, writing a plain text link is discouraged. Thus, the anti-pattern is mentioning the source but without a working link. If this is included in the description of a cloned function, making an external reference to the original package is ethical. It is possible for external pages to be modified, no longer providing information accurate to the package; in this case, not updating such link is an anti-pattern.

4.1.2 Internal

A Reference directive, dual to External reference. It can also appear in parameters and descriptions and refer to internal sources inside the same documentation package.

Discussion. It can refer to either another subsection of the documentation of the same function (e.g. redirecting to the examples) or to a general page (e.g. the documentation of a class). A working link can be provided with `\code{\link{location}}` [3]. This study found occurrences in the parameters and descriptions segments.

Example. To indicate which function produced the data to be passed as argument: `@param np Data frame returned by nuts_params()`. When indicating that internal behaviour uses a specific type of object: `@description Read simple OTU tables, mapping and taxonomy files into a \code{\link{phyloseq-class}} object`.

Anti-Patterns. Not providing a working link to an internal reference is an anti-pattern, as it may lead to misunderstandings; example (in parameters): `@param mSetObj Input the name of the created mSetObj (see InitDataObjects)`. Likewise, providing a link that does not work is also an anti-pattern.

4.2 Error

This has a single directive regarding disclosure of exception handling, and it was detected in parameters, returns and descriptions.

4.2.1 Exception Raising

Derived from the work of Monperrus et al. [1], it "states a requirement on the exceptions thrown by a method implementation". It concerns the exception that may be thrown and the situations in which it is thrown. In R, this also refers to errors printed or logged in the console.

Discussion. In R, there are no distinctions between exception types as in other languages [3]. R does have a limited exception handling throw try-catch blocks, but the developer can use `stop()` to terminate a function and display an error, `warning()` to display a problem in the console, or `message()` to show a log on the console [4]; in these cases, the functions receive a message as an argument. This directive should be disclosed when whenever these methods are used.

In many cases, when there are incompatible parameters (Restriction>Optional parameters with a Correlation to another), the developer may document the Exception Raising. It can appear in the parameters when used to indicate reattempts before erroring, e.g., `param n_retries ‘numeric‘ number of times the access to the http server should be retried in case of error before quitting.`

Anti-Patterns. Not disclosing an exception being thrown when the function does have implicit handling of exception (i.e., omitting this directive when there is an exception being raised). Likewise, stating there is an error but not providing details: `@return [...] If the sheet does not exist, return an error`.

4.3 Style

These are related to R's dynamically-typed nature and were only found in the parameters and returns.

4.3.1 Fixed

This applies to returns and parameters. It states a required property, returned by a specific function, where the type and internal structure are always the same, regardless of the function's behaviour. In a parameter, it explicitly states that an argument only receives variables of a specific type.

Discussion.

- *Return.* Because R is dynamically typed, a function can return different types on each alternative path. A fixed return implies that the return type will always be the same, but the variable's content can change. For example, if it returns a string, it will always be a string; if it returns a matrix of 2x2, it will always be the same matrix. Returning a type or `null` is still considered as a FIXED return. This is the most common return, equivalent to a traditional *return value directive* in the work of Monperrus et al. [1].
- *Parameter.* Because R is dynamically typed, the signature of a function cannot restrict the type of the arguments [3]. Developers often resort to checking type at the start of a function and erroring if it is incorrect [4]. Therefore, this primitive makes explicit that an argument can be of a single type. It should be combined with other *type directives* such as PRIMITIVE or NON-PRIMITIVE (and its variants) to provide details of the expected format (beyond the type).

Example. When a specific value is returned: `@return true if the request fails (status code 400 or above), otherwise false`. To add information about a collection's dimensions: `@return a numeric vector of the same length as x`. To clarify a specific type of custom object: `@return a ggmap object (a classed raster object with a bounding box attribute)`. To specify that the returned object is a modification of an input: `@return Modified phyloseq object`. Used without making the type explicit if the description is not ambiguous: `@param InstanceId (required); The instance IDs for which you want association status information`. Mixing the explanation with the type: `@param TP number of true positives`. Mixed with NON-PRIMITIVES: `@param object A select qdap object that stores a plot`.

Good Practices and Anti-patterns. Should be combined with TYPE, to explain the exact type of the variable being returned or passed; thus, good practices and anti-patterns of types and showcases also apply here. An R function can SHOWCASE and send a return back alongside this. Providing the type with no additional description is misleading if the variable name is an acronym or shortened name: `@param pch Numeric`. When working with numbers, the directive does not clarify the specific type (i.e. number, integer, complex): `@param eq_price The equilibrium price`.

4.3.2 Variable

This directive should be used when the type of variable to *return* is conditional: delivering different types (namely, a string and a matrix) after specific conditions. In *parameters*, this directive only appears if the parameter is not of a specific type. It is used to state that the argument can vary in type.

Discussion.

- *Return.* Understanding which return will be provided and under which circumstances is fundamental. Since the function's signature provides no clarification, the documentation should cover it. For a conditional return, at least two different types of variables are returned, i.e., a matrix and a list, or a logical and a dataframe. A `null` value will not count as conditional return. Note that R lists can be composed of any combination of types [3]. This directive only applies to lists if the inner types change, e.g., if in one case it returns a list of data-frames and in another a logical list. A list varying in size is fixed unless a list of size one is returned as the element (extracted from the list), e.g., `@return An integer, or list of integers`. As before, it can be used in combination of TYPE and SHOWCASE directives.
- *Parameters.* If the argument is always the same type but changes value, it is a FIXED directive. A VARIABLE directive can change between primitives, between non-primitive, or between them. There is no maximum limit on how many types an argument may accept, but each type's conditions and effects should be clearly stated. Suppose the argument accepts a list or vector of any size; in that case, if the single-element list can be passed as the element itself (without being in the list or vector), it is a VARIABLE directive.

Example. Combined with PRIMITIVES and NON-PRIMITIVES, to explain what each accepted type is: `@param Rho Required. Can be a single value (correlation common among all variables), a vector of the lower triangular values (vech) of a correlation matrix, or a symmetric`

`matrix of correlation coefficients`. To clarify between two NON-PRIMITIVES: `@param a a numeric vector or matrix`.

Good Practices. If an argument determines the change, the largest explanation should be in the `@param` segment, with a small reference in the return; e.g., `@return A tibble (or dataframe), or ggplot2 object if plot = TRUE`. If many types are NON-PRIMITIVES, it needs an explanation of their composition or a reference to the document that explains it. If the type of return changes according to the results of an internal calculation, it should be explained.

Anti-patterns. Not explaining the conditions upon which a specific type is returned: `@return A numerical vector or a time series object of class ts`. This is challenging since developers cannot know when or why a type of return is produced.

Not mentioning all the valid types for a *parameter*. This is acceptable in the case of the ellipsis argument, used when a function can have a variable number of arguments [4]; e.g., the argument was not an ellipsis: `@param obj A vector, matrix etc.`; the use of 'etc.' when listing types is strongly discouraged. As this type should be combined with PRIMITIVES and NON-PRIMITIVES, it is affected by their good practices and anti-patterns.

4.3.3 No Return

R functions do not have to declare a return type, so this directive clarifies that a function produces no return (neither regular nor invisible). It is the equivalent to a Java's `void` return. Thus, it only applies to the `@return` segment; note that it was placed here since it could be considered a specialisation (or particular case) of STYLE>FIXED when applied to results.

Discussion. Generally, it is not required to write the `return()` expression at the end of an R function–R returns the value of the last expression evaluated [3]. This can cause a problem if said value is a flag, an intermediate or inconsistent state, an internal value, or even unusable flags such as `NA` or `null`. This directive is used to notify that no intended return is provided and ignore it if any is automatically returned.

Example. A simple explanation with no additional comments, such as `@return None`. Or in combination with showcase: `@return None.  Results are printed`.

Good Practices. If the function produces a showcase, that additional information should be clarified. Good practices and anti-patterns related to showcases are explained in the corresponding directive and are applicable here.

4.4 Type

Like the STYLE, these are needed because R is dynamically-typed and has no reserved words to enforce types in variables. It was found only on parameters and returns.

4.4.1 Primitive

It states that the *argument* receives a primitive, or the return rebounds a primitive value. It may have several TYPE>PRIMITIVE directive kinds if it is of STYLE>VARIABLE and allows for a different primitive. It explains what the value(s) means and how it impacts the behaviour. Collections of primitive types (i.e. vectors or lists) are non-primitives. Always returning `null` can be considered a primitive return.

Discussion. The name 'primitive type' is not used in R; instead, they are defined as 'basic' [3]; this name was chosen for readability purposes. A primitive is either a character, logic, or a number (i.e., integer, numeric or complex). The explanation must mention the accepted type. In some cases, this is not required (i.e. when describing a logical). It can be used with STYLE, and alongside other TYPE directives (e.g. collection) if the return or argument is STYLE>VARIABLE. In returns, it can be used alongside SHOWCASE directives. In the parameters, it can be accompanied by a FORMAT directive.

Good Practices. Explaining the meaning; for example, `@param ntrees the number of trees in the population`. Specify if some values modify a function's behaviour: `@param num.iter Integer scalar specifying the number of iterations to use for the grid search`.

In the returns, it clarifies the conditions for each value of a logical: `@return Logical indicating whether a write occurred, invisibly`. If a character or number is returned and its structure or ranges can change according to the function's internal process, the directive should provide this information. No type requires an explicit mention to the type (e.g. 'returns a logical') if it can be safely assumed from the explanation; for instance`@return Returns (invisibly) the old root path`.

Anti-patterns. No description at all (e.g. `@param maxResults`), or a vague description that does not clarify the usage (e.g. `@param my.id Company's ID`). When a parameter has unexplained format restrictions. Not clarifying the conditions for each value when a logical is returned: `@return TRUE or FALSE`. Unclear wording that does not convey the type variable being returned: `@return 'bib' - invisibly`.

4.4.2 Non-Primitive

It indicates that a parameter or return accepts/rebounds a non-primitive type as a value. It explains what the argument's value(s) means and how it impacts the function's behaviour.

Discussion. A non-primitive is considered to be a COLLECTION (factors, lists, vectors, arrays) a DATAFRAME (matrix, dataframe, tibble, table) or an OBJECT (defined as an R object). In all of those cases, it can be accompanied of ENTRY, which details the individual values of that non-primitive. A null or `NA` return is a NON-PRIMITIVE TYPE directive if it happens variably. It can be combined with STYLE directives, or SHOWCASE directives (the latter for returns only). In the parameters, it can be accompanied of FORMAT directives such as FORMAT>ENTRY or FORMAT>SIZE.

Good Practices. Primitive values included inside the non-primitive should include the PRIMITIVE extension when specific formats are needed for a parameter. Linking to type documentation is accepted because it avoids redundancy, but any deviation from the generic document should be explained.

ENTRIES should be explained. This implies a structure of the columns (for frames), a vector's values, a list's structure, and object type. Roxygen allows creating documentation for an object; linking to said document (if it exists) instead of repeating the explanation is a good practice. For example `@return A data frame with three columns: httr (The short name used in httr), libcurl (The full name used by libcurl), and type (The type of R object that the option accepts)`. In cases like this, since Roxygen allows markdown commands, its use is encouraged.

Additional information such as encoding (if characters) or length: `@return For 'text', a character vector of length 1. The character vector is always re-encoded to UTF-8. If this encoding fails (usually because the page declares an incorrect encoding), 'content()' will return 'NA'`. If it always returns an object, but the class changes conditionally, it should be a VARIABLE directive kind (whichever corresponds), explaining the conditions in which is class is returned.

Anti-Patterns. Not providing insight into the composition of the TYPE>NON-PRIMITIVE can hinder the developers as they struggle to determine what is acceptable. Using the words 'list' or 'vector' as interchangeable can lead to misunderstandings due to differences in manipulating both types. Generic descriptions that do not clarify the type, content, or use act as an anti-pattern: `@param object object`.

Unclear wording that does not clarify the non-primitive entries without providing links; e.g., `@return A list of OAuth parameters`. Listing the entries without additional explanation about types: `@return a list containing: scheme, hostname, port, path, params, fragment, query (a list), username, password`. Lack of clarification regarding several objects being returned as a collection or as conditional returns. If a function returns a 'tibble' [3]. However, the directive describes it as a 'dataframe', the documentation is misleading since they are technically different types of data, and some functions or packages work with one but not the other; not using the proper word for a type is an anti-pattern.

4.4.3 Undefined

This is an **anti-pattern** in itself, found only on the *parameters*. It was commonly detected on the ellipsis argument (namely, `...`) when the argument was written but not used. For example, `@param ... Additional arguments, currently ignored`. Sometimes it was used generically on grouped elements, but without providing enough details to infer the types, such as `@param ... arguments passed to other methods`.

In other cases, they specified partial information, as in the following case: `@param ... Options to set, with the form name = value`. Albeit it is feasible to infer multiple arguments are passed, this is not a collection, and there are no details regarding what the names are, nor what values (types, formats) are acceptable.

4.5 Correlation

These were expanded from the work of Monperrus et al. [1]. In R, arguments are not enforced, and they can be omitted by default. As a result, often parameters are used either to change the type of a RETURN. Likewise, they can be used to alter other PARAMETERS, by using, enforcing or ignoring them (related to RESTRICTIONS), or by changing the type of value they accept (related to STYLE and TYPE).

4.5.1 Parameters

A CORRELATION directive, describing inter-dependencies involving two or more arguments [1]. The arguments must belong to the same function. This was found on both parameters and returns.

Discussion. R does not provide any syntax for cases when values are only accepted after given conditions

in another argument. This directive indicates a correlation between the parameters of the same function. Suppose the parameter is of VARIABLE, and some types can only be passed when another parameter meets a specific condition, then a PARAMETER correlation directive should be used. For a return, it is often used to indicate all types of returns and which parameters affect them.

Example. An argument can only be used if another is used: `@param longitude The longitude of observation (only used if eos="gsw"; see Details)`. One argument's characteristics are linked to another's `@param weights Numeric weights vector. Must be the same length as x`. Passing one parameter makes another compulsory: `@param scale A logical value: whether to return standard deviations or 1s. Don't use scale without using centre`. In a return, depending on an optional parameter: `@return The value of the edge attribute, or the list of all edge attributes if name is missing`.

Anti-Patterns. Not explaining the correlations, stating incomplete correlations, or not updating the correlations upon changing them. If combined with other directives, such as RESTRICTIONS, other CORRELATIONS and ERRORS, all the anti-patterns of those directives also apply here.

4.5.2 Return

A CORRELATION directive, it is a dual for PARAMETER. It describes inter-dependencies between the parameter of a method and the return it will provide. This was found only on the parameters.

Discussion. Many arguments are used to configure a function's behaviour, affecting its results. This correlation cannot be written in the method signature and must be clarified through a directive. The return of a different function may be forced as the parameter of another; in that case, the description should also have a STATE>SEQUENCE directive.

Example. To affect how many (and which) results will be returned: `@param nextToken The token for the next set of items to return. (You received this token from a previous call)`. To affect showcase returns (i.e. printing or plotting): `@param silent keep output as silent as possible. Defaults to true`.

Anti-Patterns. Stating incomplete correlations or not updating them after a change. Not mentioning the correlation on the return segment of the documentation. If combined with other directives, such as RESTRICTIONS, other CORRELATIONS and ERRORS, all the anti-patterns of those directives also apply here.

References

[1] M. Monperrus, M. Eichberg, E. Tekes, and M. Mezini, "What should developers be aware of? an empirical study on the directives of api documentation," *Empirical Software Engineering*, vol. 17, no. 6, pp. 703–737, Dec 2012.

[2] D. Budgen, *Software Design*, 2nd ed. USA: Addison-Wesley Longman Publishing Co., Inc., 2003.

[3] H. Wickham and G. Grolemund, *R for Data Science: Import, Tidy, Transform, Visualize, and Model Data*, 1st ed. USA: O'Reilly Media, Inc., 2017.

[4] H. Wickham, *Advanced R*, ser. Chapman & Hall, CRC The R Series. Boca Raton, Florida: CRC Press, 2015.

[5] G. Zanella and C. Z. Liu, "A Social Network Perspective on the Success of Open Source Software: The Case of R Packages," in *Hawaii International Conference on System Sciences*. HAwaii: HICSS, Jan. 2020, pp. 471–480.

[6] K. Li, P.-Y. Chen, and E. Yan, "Challenges of measuring software impact through citations: An examination of the lme4 R package," *Journal of Informetrics*, vol. 13, no. 1, pp. 449–461, Feb. 2019.

[7] A. Turcotte and J. Vitek, "Towards a Type System for R," in *Proceedings of the 14th Workshop on Implementation, Compilation, Optimization of Object-Oriented Languages, Programs and Systems*, ser. ICOOOLPS '19. London, United Kingdom: Association for Computing Machinery, Jul. 2019, pp. 1–5.

[8] K. Hornik, "Are there too many r packages?" *Austrian Journal of Statistics*, vol. 41, no. 1, pp. 59–66, 2012.

[9] W. Cho, Y. Lim, H. Lee, M. K. Varma, M. Lee, and E. Choi, "Big Data Analysis with Interactive Visualization using R packages," in *Proceedings of the 2014 International Conference on Big Data Science and Computing*, ser. BigDataScience '14. Beijing, China: Association for Computing Machinery, Aug. 2014, pp. 1–6.

[10] C. Maddumage and M. P. Dhanushika, "R programming for Social Network Analysis - A Review," in *2018 3rd International Conference on Information Technology Research (ICITR)*. Moratuwa, Sri Lanka: IEEE, Dec. 2018, pp. 1–5.

[11] A. Perperoglou, W. Sauerbrei, M. Abrahamowicz, and M. Schmid, "A review of spline function procedures in R," *BMC Medical Research Methodology*, vol. 19, no. 1, p. 46, Mar. 2019.

[12] M. Claes, T. Mens, N. Tabout, and P. Grosjean, "An empirical study of identical function clones in CRAN," in *2015 IEEE 9th International Workshop on Software Clones (IWSC)*. Montreal, Canada: IEEE, Mar. 2015, pp. 19–25.