Paper CT10

risk.assessr: extracting OOP function details

Edward Gillian, Cytel, Gorzów Wielkopolski, Poland

Hugo Bottois, Sanofi, Paris, France

Paulin Charliquart, Sanofi, Stockholm, Sweden

Andre Couturier, Sanofi, New Jersey, USA

ABSTRACT

Extracting details from Object Oriented Programming (OOP) functions in R has proved challenging as there are different types of these functions (S3, S4, R6, special) which have different methods for encoding and diverging standards for writing them. Firstly, to extract details, these functions need to be identified in the NAMESPACE and S3 functions have different NAMESPACE constructs to other OOP functions. These functions can be constructed with different and unique classes that must be checked. These functions can be documented in diverging ways using usage descriptors such as “"defunct|deprecated|deprec|superseded|imported|re-exported|reexports". Secondly, these different function types require considerably different methods to extract the details ranging from functions called from the methods package and base R to custom functions to create extraction inputs.
These diverging encoding methods and standards had implications for creating code that can be executed in various processes. For GitHub CICD chains, considerably more error handling had to be programmed to allow the CICD chains to execute. For creating a more accurate traceability matrix, additional coding to look for the usage descriptors needed to be added. For evaluating if functions called functions from Suggested dependencies, additional coding was needed to process OOP functions that had special naming conventions and classes.

INTRODUCTION:

Extracting details from Object Oriented Programming (OOP) functions in R has proved challenging for a number of reasons. There are different types of these functions (S3, S4, R6, special) and they have different approaches encoding within the R package structure for access, naming them, and for writing them. Specifically, the different function types can have different visibilities or registrations within R packages. With these different registrations, these functions need different methods to access them. The major method to access them is through the NAMESPACE as they can be identified easily in this file. However, not all R OOP functions reside directly in the NAMESPACE (e.g. R6 functions). Some OOP functions can be accessed through other OOP function types (e.g. S3 Methods can instantiate or interact with R6 Internals). At times, these functions can be identified by means of different and unique classes. These different visibilities mean that extracting function details can be executed by a specific sequence of processing based on the ease of detection and popularity of use.

The purposes of extracting details about OOP functions were to identify functions with risks for reporting in traceability matrices and checking if functions in the target package call functions from Suggested dependencies.

Function types and impact on encoding in R package structures

In R, the function type and its visibility, whether they are exported, internal, or accessible through other functions and/or namespaces, depends on how they are defined and registered within a package. These registrations can be particularly complex and overlap when dealing with S3 methods, R6 classes, and R6 class generators. The next section discusses the reasons for gathering information on R package functions.

Purpose for gathering details for functions in R packages

The Sanofi validation team requires information about R package functions that could be at risk for use in submissions to regulatory authorities. To do so, the team has gathered data on function details such as:

- unit test coverage,
- function status (e.g. deprecated, superseded, experimental),
- function class (e.g. regular function, S3, S4, S7, R6),
- function description (roxygen2 documentation),
- source body code.

To that end, firstly, the functions need to be identified through their type and class (if applicable). Once the function type information is identified, the data gathered are stored in a Traceability Matrix Object (see section "Outline of the process sequencing in classify_function_body()" for details about how the data is stored and reported) that presents the information in a table/matrix format. This allows the team to link all associated tests/documentation files/R scripts with their associated R package functions. This traceability matrix allows the team to identify if functions have code using packages in the Suggests section and in the future, execute function comparison across package versions.

Once the Sanofi validation team has information about the functions, the team can make informed decisions about how to assess and mitigate the risk for a specific function. The following section discusses the major function types in R and how those types impact their encoding in R packages.

Exported vs. Non-Exported Functions

The most used function types in R packages are exported functions and internal functions (LaZerte, 2021). Exported functions are the most visible. Internal functions are less visible and are not meant to be user facing. The following table summarizes the major features of exported vs. non-exported functions.

Function Type	Features	examples
Exported functions	explicitly listed in a package's NAMESPACE file using the export() directive.	Marked with @export in the function documentation
	directly accessible via the package name	mypackage::myfunction() - access functions using double colon
	typically include user-facing functions, S3 generics, and constructors for R6 classes.	
Non-exported functions	defined in the package but not listed in the NAMESPACE file	Marked with @keywords internal and sometimes @Rd (no documentation generated)
	accessible only via introspection tools	getAnywhere() - looks at all loaded namespaces, whether or not they are associated with a package on the search list
	can include helper functions, internal methods, or S3 methods not intended for direct use	mypackage:::myfunction() - access functions using triple colon

Table 1: Exported vs. non-exported functions

S3 Method registration and visibility

S3 methods in R employ the naming convention of generic.class. These methods are registered using the S3method() function for visibility in the NAMESPACE file of a package. This registration allows R to correctly transmit methods for specific classes when a generic function is called. The S3method(generic, class) function registers a function named generic.class, while S3method(generic, class, function) allows registration of a differently named function for namespace usage (R Core Team, S3method: Register S3 Methods, 2025).

The visibility of a S3 method depends on whether the method is exported to the namespace. If explicitly exported, it is accessible via mypackage::generic.class. Non-Exported S3 Methods are not directly accessible unless using getS3method() or getAnywhere(). S3 methods can reside in the namespace even if not exported, and they can be discovered using the following functions:

getS3method("print", "myclass")

getAnywhere("print.myclass")

R6 Class Generators vs. R6 objects

R6 Class Generators

R6 Class Generators are created through R6::R6Class(). They are the method for creating R6 objects. The generator contains the following metadata:

R6 metadata	usage
$public_methods	methods accessible to users.
$private_methods	internal methods.
$classname	name of the class.
$inherit	inheritance from another R6 class.

Table 2: R6 class generators

These class generators are usually exported if the class is meant for public use. In the Shiny package (Chang W. C., 2025), the reactiveValues class is an exported R6 class generator (van Leemput, 2024). Internally, reactiveValues is created using R6::R6Class() and exported via the NAMESPACE file, so you can access it directly using shiny::reactiveValues. Table 3 shows a code sample using reactiveValues:

```
library(shiny)

# Create a reactiveValues object

rv <- reactiveValues(counter = 0)


# Access and modify fields

rv$counter <- rv$counter + 1

print(rv$counter)  # 1
```

Table 3: reactiveValues code sample

On the surface, reactiveValues() returns an R6 object which can be accessed and the fields can be modified using $. The object is reactive, meaning changes to its fields can trigger updates in Shiny UI components. Behind the scenes, programmers use reactiveValues() like a function, but the function actually returns an instance of an R6 class (van Leemput, 2024).

R6 Objects

These are created by calling $new or updating the value in the function as in Table 3. The object contains methods and fields bound to the class and the methods are accessed through the object. Important points to note are method definitions are inspected through the generator which has a list-like structure and the object is an instance with a state.

Functions with Multiple Types

Functions in R can be defined as more than one type depending on context. The following provides an example of this:

Generic Functions: These function types can dispatch to S3, S4, or R6 methods depending on the object class (R Core Team, Methods for S3 and S4 Dispatch, 2025). A generic function example is as.list(). It can dispatch through:

Dispatch method	Comment
S3	as.list.data.frame, as.list.factor, etc.
S4	setMethod("as.list", "MyS4Class", ...)
R6	Can define as.list method manually.

Table 4: Generic function as.list dispatch methods

Sequence of Function type processing

The variety of OOP and other function types in R and their differing registration and visibility in an R package structure mean that processing functions for their details needs to take place in a particular sequence to be most effective in capturing those elements. In our R package for validation, risk.assessr (Sanofi, 2025), two functions: extract_exported_function_info() and classify_function_body() were written to get the names of functions and extract function details from those functions. The function extract_exported_function_info() is a utility designed to extract metadata about exported functions from an R package. It classifies these functions into categories such as regular functions, S3, S4, and S7 methods or classes. The function classify_function_body() has a specific sequence of processing based on checking functions and their explicit registration, ease of detection, popularity of use, and interaction with other function types.

Explanation of extract_exported_function_info()

This utility function retrieves and analyzes the namespace of a given R package and extracts its exported functions' metadata. It searches for regular functions and R's exported OOP functions (S3, S4, and S7).

The function starts with parsing the package's namespace using parseNamespaceFile() from base R. This function reads the NAMESPACE file. Its output is a structured list containing exports, S3methods, and exportMethods (R Core Team, Writing R Extensions, 2025). This method may not retrieve all the exported functions as the NAMESPACE file may not contain all the entries or be malformed. As a result, the function includes a fallback function getNamespaceExports(), which retrieves the exports from a loaded namespace environment (R Core Team, R Data Import/Export, 2025). This dual approach is necessary as parseNamespaceFile() works with the source code and not the loaded package. On the other hand, getNamespaceExports() requires a loaded target package and shows actual runtime exports, which may differ from the data that parseNamespaceFile() provides due to dynamic registration or conditional logic in .onLoad() hooks (R Core Team, Full Reference Manual for R Version 4.5.1, 2025).

From this dual approach output, the function uses getExportedValue() to extract the actual object associated with each name in the exported function data object (Wickham H. , Managing Imports and Exports, 2022). Then, the object's class and structure are checked. The classification logic of these two parameters includes the following checks:

Exported OOP type	Method of identification
S7 generics and classes	identified using inherits(obj, "S7_generic") and inherits(obj, "S7_class").
S7 constructors	identified via pattern matching in the function body for calls to S7:: new() or S7::new_object()
Regular functions	default classification for callable objects not matching other criteria.

Table 5: Exported OOP type function identification

This function collects further data on S3 and S4 methods. S3 methods are extracted from the S3methods field in the namespace list. These are typically registered using S3method(generic, class) function in the NAMESPACE file (Bengtsson, 2022). S4 methods and generics are retrieved from exportMethods, which may contain either character vectors or structured lists. These methods are registered using exportMethods() or exportClass() functions (Wickham H. , sloop: Helpers for 'OOP' in R, 2019).

This multiple method approach tries to ensure that exported methods registered through different means are captured accurately. Notably, S3 methods may not be directly accessible unless explicitly exported, and S4 methods require formal registration to be discoverable.

Justification for Multiple Methods

Multiple methods of extracting exported functions are needed as R namespace files can be incomplete or dynamic. Static parsing and extraction may miss dynamically registered methods. Also, there can be discrepancies at runtime as some OOP methods are only available after the package is loaded and initialized. Next, with R supporting multiple OOP systems (S3, S4, and S7) and these systems each have distinct registration and dispatch methods. Lastly, employing both static and dynamic inspection increases the extraction reliability across diverse package structures.

Explanation of classify_function_body()

This function in R is a utility that further classifies the function types in a package and extracts the function body. It executes on a data frame transformed into a list row by row of metadata by using an apply function and produces a transformed data frame. The function handles various R OOP systems (S3, S4, S7, R6, and ggproto) by extracting the function body and classifying its type accordingly.

The function adheres to a top-down approach based on the following criteria:

Processing criteria	Comments
Explicit registration and visibility	S4 generics are registered - easier to detect (Genolini, 2008).
	S3 methods are inferred from naming and method tables (Bengtsson, 2022).
Interaction with other function types	S7 classes are structured - detectable via inheritance (Wickham H. R.-O., 2023).
	S3 methods may wrap R6 or ggproto - check S3 first.
	S7 generics may wrap S4 - check S4 first.
Fallback safety	Regular functions and numeric values are checked last, as they are least likely to be mistaken for OOP constructs.

Table 6: Processing criteria

Outline of the process sequencing in classify_function_body()

This function determines how a target function should be interpreted based on its structure and context and employs a particular sequence of processing function types in aid of this goal. The sequence focusses on function types, visibility, generic/class names, and interactions between OOP systems. The sequence is laid out in Table 7:

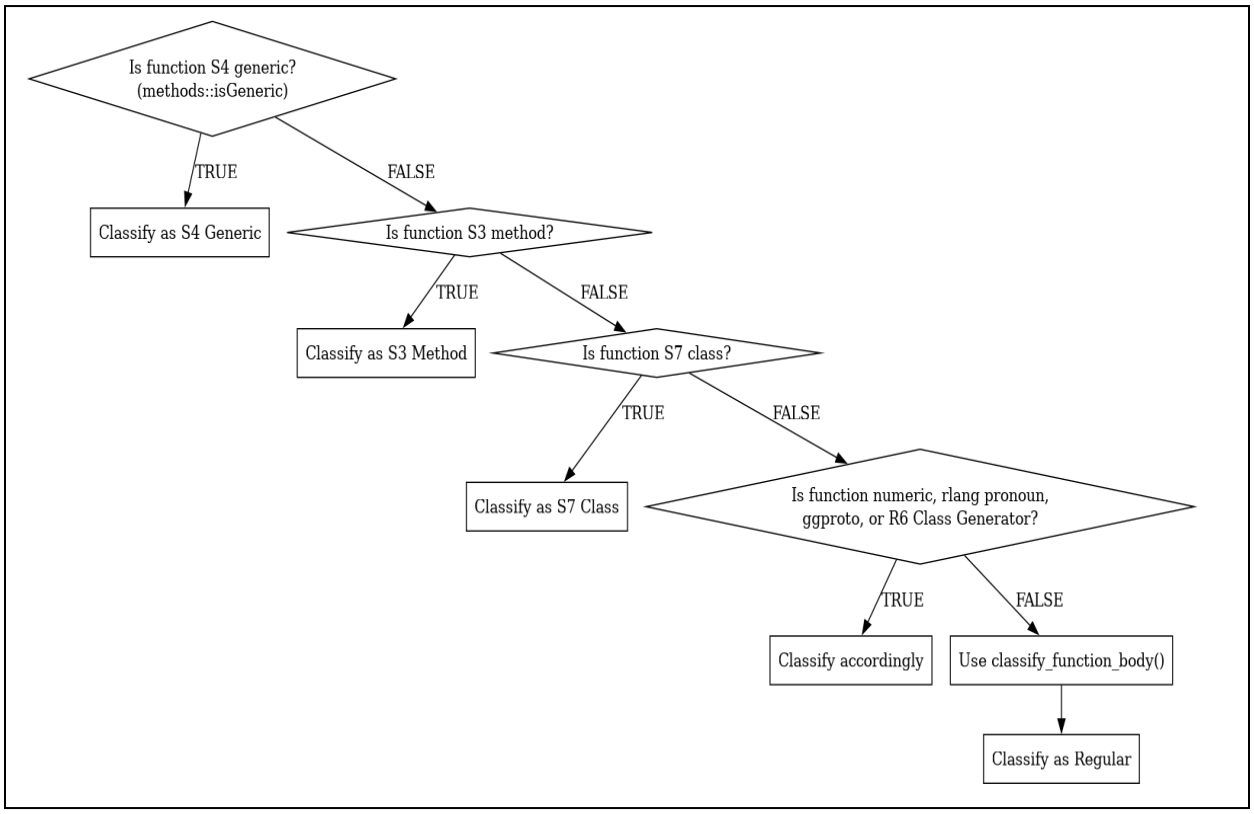

Table 7: Processing sequence

S4 generic (Genolini, 2008) are processed before S7 generic as they are explicitly declared through using setGeneric() and registered in the methods table, making them easily detectable via methods::isGeneric(). This makes them easier to identify early in the classification process. An example from teal.code version 0.5.0 (Kaledkowski, 2025) is the get_code() function. In Sanofi's traceability matrix, this is identified as a S4 generic and method (see Table 8 for more details).

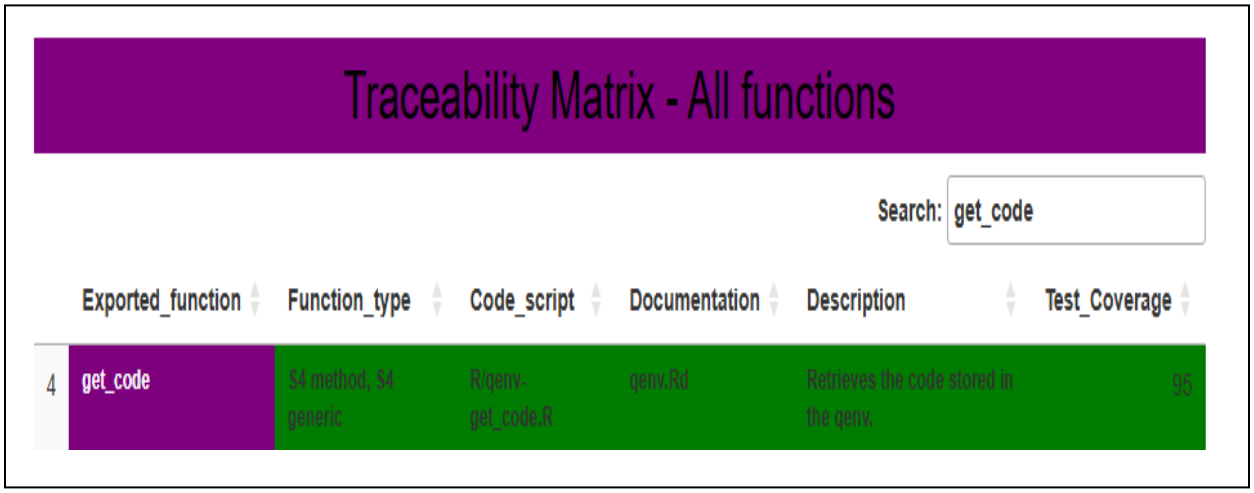

Table 8: get_code() – Sanofi traceability matrix

S4 has been part of base R since 2005 unlike S7 (Wickham H. R.-O., 2023) and implemented through the methods package (R Core Team, methods: Formal Methods and Classes (Version 4.5.1), 2025) . By prioritizing S4 in the sequence, this ensures compatibility with legacy and widely used packages. Finally, S7 borrows concepts from S4, including generics and method dispatch. If a function is both an S4 and S7 generic, it is more reliable to detect S4 first, since S7 may delegate to S4 internally.

Next in the sequence is S3 Method. These are processed ahead of S7 class and other types as S3 is the most common OOP system in R. An example from R6 version 2.5.1 (Chang W. , 2025) is the generic function as.list(). In Sanofi's traceability matrix, it is identified as a S3 generic and S3 method (see Table 9 for more details).

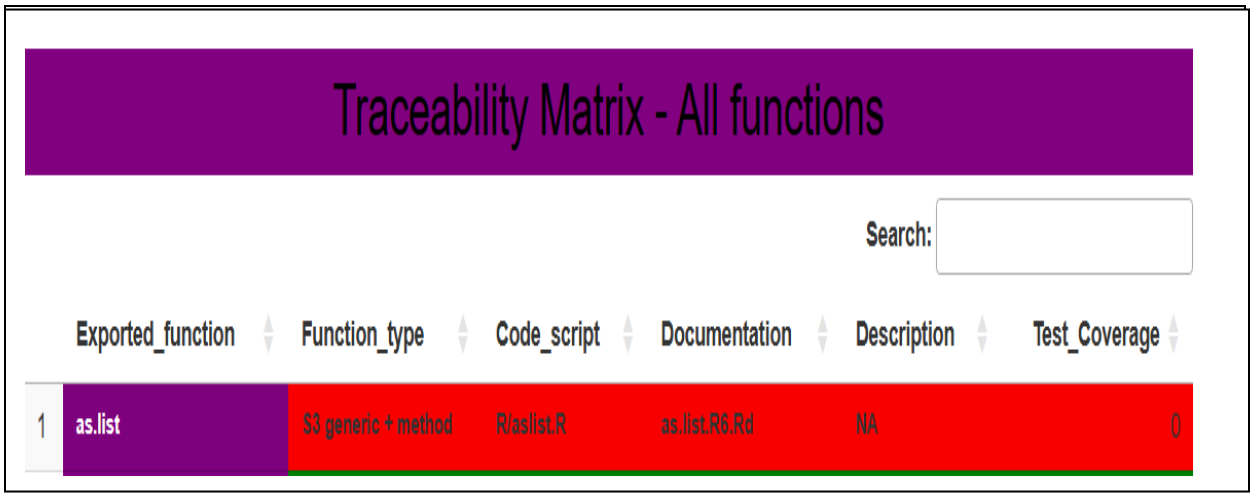

Table 9: as_list() – Sanofi traceability matrix

When examining the R6 NAMESPACE file, it is registered as an S3 method but is a generic as well. S3 methods often instantiate or manipulate R6 objects. In the R6 version 2.5.1 NAMESPACE, the generic print is an S3 method with a class of R6 generator. Detecting S3 early in the processing sequence allows the function to be correctly classified as a method, even if it uses R6 internally. They are less likely to be mistaken for S3 methods, so their processing can be delayed.

There are other types of functions considered in the processing sequence, including numeric, rlang pronoun, ggproto, and R6 Class Generator.

Finally, if a function cannot be matched to any of the above types, then classify_function_body() attempts to extract the body details and assigns the class as "regular".

Discussion of the processing sequence

The classify_function_body() function implements a prioritized sequence for processing different function types, focusing on their detectability, popularity, and interoperability. This approach reflects a balance between technical detectability and practical usage patterns, to allow for robust and comprehensive metadata extraction across diverse R packages. The function also considers in the sequencing that R OOP systems can be interoperable. S3 methods can instantiate or manipulate R6 objects, especially in packages that use R6 for internal state but expose S3 interfaces. Some packages use S3 methods for backward compatibility while defining S4 generics. S7 is designed to unify S3/S4, so it may wrap or mimic behaviors from both.

CONCLUSION

The extraction of OOP function details in R is complex due to the number of OOP systems (S3, S4, R6, and S7) and their varying mechanisms for registration, visibility, and interaction. Each system introduces its own conventions for naming, exporting, and structuring functions, which in turn affects how these functions can be programmatically discovered and analyzed.

While the NAMESPACE file serves as a primary entry point for identifying exported functions, it does not capture all OOP constructs; particularly, those from systems like R6, which rely on runtime instantiation rather than static registration. Moreover, the interplay between different OOP systems (e.g. S3 methods invoking R6 internals) further complicates the extraction process.

Given these challenges, a layered and adaptive approach is essential. This involves prioritizing detection based on visibility and registration clarity; starting with explicitly registered S4 and S7 functions, followed by S3 methods inferred from naming conventions, and finally R6 components that require inspection of class definitions. Such a strategy ensures both completeness and accuracy in extracting function metadata, supporting robust tooling for documentation, testing, and static analysis in R package development.

CONTACT INFORMATION

Your comments and questions are valued and encouraged. Contact the author at:
Author Name: Edward Gillian
Company: Cytel

Address: 675 Massachusetts Ave Cambridge, MA 02139 USA
Work Phone: +1 (617) 661-2011
Email: edward.gilllian-ext@sanofi.com
Website: www.cytel.com

References

Bengtsson, H. (2022). *R.methodsS3: S3 Methods Simplified.* Retrieved from The Comprehensive R Archive Network (CRAN): https://cran.r-project.org/web/packages/R.methodsS3/R.methodsS3.pdf

Chang, W. (2025, September 26). *R6: Encapsulated Classes with Reference Semantics. R package version 2.5.1,*. Retrieved from https://github.com/: https://github.com/r-lib/R6

Chang, W. C. (2025, September 25). *shiny: Web Application Framework for R*. Retrieved from The Comprehensive R Archive Network (CRAN): https://cran.r-project.org/package=shiny

Genolini, C. (2008). *A (Not So) Short Introduction to S4 Object Oriented Programming in R (Version 0.5.1).* Retrieved from The Comprehensive R Archive Network (CRAN): https://cran.r-project.org/doc/contrib/Genolini-S4tutorialV0-5en.pdf

Kaledkowski, D. C. (2025, September 26). *teal.code: Code Storage and Execution Class for 'teal' Applications. R package version 0.5.0,*. Retrieved from https://insightsengineering.github.io/teal.code/

LaZerte, S. (2021, November 16). *How to cite R and R packages.* . Retrieved from rOpenSci.: https://ropensci.org/blog/2021/11/16/how-to-cite-r-and-r-packages/

R Core Team. (2025, September 25). *Full Reference Manual for R Version 4.5.1.* Retrieved from The Comprehensive R Archive Network (CRAN): https://cran.r-project.org/doc/manuals/r-release/fullrefman.pdf

R Core Team. (2025, September 25). *Methods for S3 and S4 Dispatch*. Retrieved from rdrr.io: https://rdrr.io/r/methods/Methods_for_S3.html1

R Core Team. (2025, September 25). *methods: Formal Methods and Classes (Version 4.5.1).* Retrieved from The Comprehensive R Archive Network (CRAN): https://cran.r-project.org/doc/manuals/r-patched/packages/methods/refman/methods.html

R Core Team. (2025, September 25). *R Data Import/Export*. Retrieved from The Comprehensive R Archive Network (CRAN): https://cran.r-project.org/doc/manuals/r-release/R-data.html

R Core Team. (2025, September 25). *S3method: Register S3 Methods*. Retrieved from rdrr.io: https://rdrr.io/r/base/S3method.html

R Core Team. (2025, September 25). *Writing R Extensions*. Retrieved from The Comprehensive R Archive Network (CRAN): https://cran.r-project.org/doc/manuals/R-exts.html

Sanofi. (2025, February 6). *risk.assessr*. Retrieved from risk.assessr: https://github.com/Sanofi-Public/risk.assessr

van Leemput, V. (2024, October 24). Retrieved from Shiny source code explained: The use of R6: https://hypebright.nl/en/shiny-en/shiny-source-code-explained-the-use-of-r6-2/

Wickham, H. (2019). *sloop: Helpers for 'OOP' in R.* Retrieved from The Comprehensive R Archive Network (CRAN): https://cran.r-project.org/web/packages/sloop/sloop.pdf

Wickham, H. (2022). *Managing Imports and Exports.* Retrieved from The Comprehensive R Archive Network (CRAN): https://cran.r-project.org/web/packages/roxygen2/vignettes/namespace.html

Wickham, H. R.-O. (2023). *Classes and Objects in S7.* Retrieved from The Comprehensive R Archive Network (CRAN): https://cran.r-project.org/web/packages/S7/vignettes/classes-objects.html

ACKNOWLEDGMENTS

Edward Gillian and Hugo Bottois participated in providing and implementing the technical solutions. Andre Couturier handled project management and guided the project direction. Paulin Charliquart conducted the project and documentation review.