SoftwareX 19 (2022) 101113

Contents lists available at ScienceDirect

SoftwareX

journal homepage: www.elsevier.com/locate/softx

Original software publication

analyzeR: A SonarQube plugin for analyzing object-oriented R Packages

Pranav Chandramouli [a], Zadia Codabux [a], Melina Vidoni [b,*]

[a] University of Saskatchewan, Department of Computer Science, 176 Thorvaldson Bldg, Saskatoon, Canada
[b] Australian National University, CECS School of Computing, 108 North Road, Canberra, Australia

ARTICLE INFO

Article history:
Received 7 March 2022
Received in revised form 3 May 2022
Accepted 19 May 2022

Keywords:
SonarQube
R packages
Static code analysis
Object-oriented
R plugin

ABSTRACT

Automated Static Analysis Tools (ASATs) analyze source-code to capture defects and ensure higher quality. SonarQube is a renown ASAT that supports mainstream programming languages. However, R programming is not included. R is an increasingly popular multi-paradigm and package-based programming environment for scientific programming. Nevertheless, R's Object-Oriented (OO) functionalities are implemented through three different systems: S3, S4, and R6, and seldom used by developers. We present **analyzeR**, an advanced SonarQube plugin to examine R packages built in any of the current OO models. It implements widely-used, commonly-accepted OO metrics and displays the results using SonarQube's graphical interface for increased usability, implementing an array of metrics.

© 2022 The Author(s). Published by Elsevier B.V. This is an open access article under the CC BY-NC-ND license (http://creativecommons.org/licenses/by-nc-nd/4.0/).

Code metadata

Current code version	1.0.0
Code repository	https://github.com/ElsevierSoftwareX/SOFTX-D-22-00060
Legal code license	MIT
Code versioning system used	git
Software code languages, tools, and services used	R, Python, Java
Compilation requirements, operating environments, and dependencies	SDKs: Python, Java, R. SonarQube seerver available.
Link to developer documentation	https://github.com/tdresearchgroup/analyzeR-SonarQubePlugin/wiki
Support email for questions	melinavidoni@ieee.org

Software metadata

Current software version	1.0.0
Permanent link to executables of this version	https://github.com/tdresearchgroup/analyzeR-SonarQubePlugin/releases/tag/release
Legal Software License	MIT
Computing platforms/Operating Systems	Windows/Linux
Installation requirements & dependencies	R, Python, Java. SonarQube server.
User Manual	https://github.com/tdresearchgroup/analyzeR-SonarQubePlugin/wiki
Support email for questions	melinavidoni@ieee.org

1. Motivation and significance

Automated Static Analysis Tools (ASATs) analyze source-code to diagnose important shortcomings, including Technical Debt (TD) [1,2], and are usually incorporated in the development process through Continuous Integration (CI) [3]. SonarQube[1] is a widespread ASATs tools, which [has been adopted by over 85000 organizations world-wide, and is] included in over 15000 public open-source projects as a requirement for contribution [4].

* Corresponding author.
E-mail addresses: zcodabux@cs.usask.ca (Zadia Codabux), melina.vidoni@anu.edu.au (Melina Vidoni).

1 https://www.sonarqube.org.

https://doi.org/10.1016/j.softx.2022.101113
2352-7110/© 2022 The Author(s). Published by Elsevier B.V. This is an open access article under the CC BY-NC-ND license (http://creativecommons.org/licenses/by-nc-nd/4.0/).

It supports over 25 programming languages, and checks code compliance against a set of coding rules, allowing adding custom rules [5]. In particular, "if the code violates any of the classified rules, SonarQube considers it as a violation or a Technical Debt item" [6].

TD refers to the consequences of taking shortcuts and making poor design choices during the development of a software system, negatively impacting its future evolution [7], and reflects the cost of additional rework caused by choosing an easy solution instead of a better, longer approach [8]. Developers often rush to complete tasks for various reasons, such as cost reduction, short deadlines, or even lack of knowledge [9]. This is true even though scientific software aims at understanding a problem rather than obtaining commercial benefits [10,11].

TD's impact on scientific software can be more damaging since its *special purpose* is to process research results in many disciplines [10,12]. This happens because "part of the complexity in measuring the scientific software ecosystem comes from the way that different pieces of software are brought together and recombined into workflows and assemblies" [11]. New research software 'runs off' previous software (i.e., packages), and its maintainability is affected by the packages it relies on [13]. Scientific software has other characteristics increasing the impact of TD: (a) research software engineers are versed in the domain (i.e., science) and will become end-users of that software [14], (b) they seldom conduct an elicitation process and organically design the software [15], (c) unit testing is often of low-quality [16], and (d) there is a gap between Software Engineering (SE) and scientific programming, which risks the production of reliable scientific results [17].

R gained importance as a popular programming language for data-science, and ranked 7th in popularity in 2021.[2] R has an ever-growing community, but most package-contributors are not software engineers by trade, and only a few apply sound development practices [14]. Previous studies demonstrated a lack of SE research for R [15], and even organizations peer-reviewing R packages still rely on incomplete ASATs to complement human reviewers' work [18].

Nowadays, research on TD in scientific software and R programming is becoming popular [15,16,18–21]. However, the availability of ASATs tools for R remains limited (Section 1.1). Because R is multi-paradigm, current tools focus on the functional-paradigm aspects or provide partial support for some types of R's OO implementations. This is problematic because ASATs' popularity of tools, and of SonarQube, is increasing rapidly [4] since ASATs are adopted to measure software quality and TD [22].

Therefore, **analyzeR** was designed to assist with static code analysis on three types of R's OO implementations (S3, S4, and R6) while maintaining a seamless, unique and user-friendly process. It is an expansible SonarQube plugin that can be easily adapted into existing CI environments, and its results can be compared to those offered for other programming languages.

1.1. Related work

TD and SonarQube. When compared to other TD-detection tools, SonarQube and CAST are the most popular [23]. SonarQube and Ptidej were used to curate a dataset of TD issues from 33 Apache Java projects [24]. The Python Apache environment was also analyzed through SonarQube, determining that more than half of Python's TD is short-term, repaid in less than two months, with only a minority of rules addressed [25]. In another study, over 17k commits of 20 Python Apache projects were analyzed with SonarQube, demonstrating that two-thirds are self-fixed at a rate negatively correlated with the number of commits, developers and project size [26]. The same dataset was used to examine co-occurring TD, uncovering that co-occurring issues are hard to remove [27].

Others studied the accuracy of SonarQube's estimated remediation time in Java when compared to junior developers' repayment time [6,22]. Similar studies compared SonarQube's predictions to regression modeling techniques [28,29]. Regardless of the language, junior developers value SonarQube's composite quality indicators, even if they do not fully understand their meaning [30].

SonarQube was used to study how TD's amount, composition and history, changed during the development of complex, open-source software through a small but language-diverse sample [31]; they determined early-versions have more TD than late-versions and that extensive refactoring reduces TD levels. SonarQube was combined with Arcan to study Architectural Debt and opportunistic code reuse in Java projects, determining that 'cyclic dependency' is a typical smell [32]. SonarQube was also applied to StackOverflow's Java code snippets, determining that reused code tends to exhibit "a substantially lower TD density" [33]. A large-scale analysis based on GitHub's annual report analyzed SonarQube's coding violations and mapped them to the most common code smells by Fowler [34]; this enabled estimating developers' profiles according to coding maturity and TD tolerance, among other points.

ASATs in R Programming. `covr`[3] is the de-facto tool for unit-testing coverage analysis, used in Test TD studies [16], and provides functionalities to manually transfer some analyses to SonarQube. `goodpractice`[4] statically investigates R packages, and is used in the peer-review process at rOpenSci [18]; it provides mostly stylistic critique (e.g., not having more than 80 characters per row) and other R-specific format checks. Regarding ASATs, `lintr`[5] is a popular tool, providing analyses limited to code. `lintr` must be run from R after an extensive configuration and can only be linked to GitHub Actions or Travis CI.

`SonarRPlugin` was the first SonarQube plugin to examine R Packages.[6] It remains on an alpha state and does not provide syntax highlight, code coverage, or statistics, and its few analysis are extracted from `lintr`. This plugin only works on S4 and does not include the recently developed R6. Therefore, contributing to this project was not feasible in the short term, as our **analyzeR** is based on a different architecture, meant to be sustainable over time and to keep including future OO systems.

2. Software description

analyzeR[7] is a SonarQube plugin to perform static code analysis on R packages' written with any of R's OO approaches.

2.1. Object-oriented programming in R

R is dynamically typed, vectorized, both lazy and side-effecting, fostering functional and interactive programming but also providing core OO features [35]. Although OO is available, it is considered advanced programming in R [36], and not approached by many users [37]. Regarding R's OO capabilities, it is worth noting that "most objects in R are not themselves reference objects" but instead "based on a concept of local references; that is, reassigning part of an object referred to by name alters the object

2 https://spectrum.ieee.org/top-programming-languages-2021.

3 https://github.com/r-lib/covr.

4 https://github.com/MangoTheCat/goodpractice.

5 https://github.com/r-lib/lintr.

6 https://github.com/Merck/sonar-r-plugin.

7 https://github.com/tdresearchgroup/R-Plugin.

referred to by that name, but only in the local environment'' [38]. R's OO implementation is based on multiple systems available to choose from; Reference Classes, S3 and S4 (all provided by base R), and the newly released R6 [36].

S3 was implemented near 1990 and introduced class attributes that allowed single-argument methods. S3 has no formal definition for an object, and attributes/classes can be changed manually, as their implementation is similar to objects in prototype-based languages (e.g., JavaScript) [39]. S3 is minimalist and flexible, allowing TD introduction [36], but is the only OO-implementation used in the packages distributed with base R [39].

S4 provides a formal approach to functioning OO in R, and its stricter implementation makes use of specialized functions for creating classes [36]. Therefore, S4 is closer to traditional OO concepts of stricter languages (i.e., Java) [39], being more verbose [38]. Particularly, BioConductor[8] (a user-led organization encapsulating tightly-integrated R packages) is based in S4.[9]

Reference Classes (RC) are sometimes called S5 in R's lingo. RC objects are mutable S4 objects, which perform more efficiently than S3/S4 [36]; RC's use is not as widespread [39]. RC was used to develop the new R6, which is installed as an R package [40]. R6 allows building fields and variables in separate environments, while also allowing for visibility (public and private methods), active bindings and inheritance [36]. R6 is reportedly better performing than other R's OO implementations [40].

`R.oo` [41] allowed S3 objects to become mutable, but its latest development efforts were completed in 2019. `proto` was another package for prototype programming, used for both class-free and class-based OO, but its latest version available is from 2016, and its original repository has been removed [42].

2.2. SonarQube architecture

SonarQube is an ASAT to detect bugs, vulnerabilities, and code smells in the code [1]. In a typical development process, developers implement and merge code, and SonarQube-integrated CI assesses the code; its 'Scanner' component posts results to a server [5]. SonarQube's architecture is composed of three main components: the *web-server* to handles SonarQube configuration, the *search server* (based on ElasticSearch) to supports searches from the server, and the *compute engine* to tally reports and metrics.

SonarQube plugins are installed onto the server.[10] Plugins extend SonarQube support to new languages, add custom metrics, improve integration, and add visualizations [4]. However, plugins run in isolated classloaders, allowing the use of third-party libraries without any risk of runtime conflicts with internal SonarQube libraries or other plugins [5]. Once the plugin is correctly appended, the 'Scanner' analyzes the source files. If the traditional scanner does not support all required file types, SonarScanner[11] must be used to scan source-code instead.

In the officially-supported plugins, SonarQube measures and grades code with traditional metrics arranged in categories [5]: *Complexity* (only including cyclomatic complexity calculated by the number of paths through the code), *Duplication* (related to duplicated blocks, files, and lines), *Issues* (problems by severity raised in the code), *Maintainability* (calculation of detected code smells, and TD ratio,[12] *Quality Gates* (existing warnings and alerts), *Reliability*, (bug-counts and estimated remediation efforts), *Security* (different types of vulnerabilities), *Size* (general metrics related to code size), and *Tests* (unit testing coverage and related metrics). SonarQube's database stores these metrics, allowing developers to track changes over time.

SonarQube promotes an approach called 'Clean as Your Code' [5] through QualityGates.[13] It allows users focus on new code, ensuring it is clean and safe. These terms are defined according to the user's standards and how they configured the status—ideally, a 'green' status means the gates (i.e., the quality standard) have been 'passed'. Quality Gates also examine pull-requests to ensure the code meets the user's standards and specifications before merging. It allows the user to see a pull-request's Quality Gates within the SonarQube visual interface, which can then be used to 'decorate' pull-requests with SonarQube issues by adding metadata, as in the 'decorator' pattern.

2.3. analyzeR architecture

The **analyzeR** Plugin is divided into three main components working through a pipe-and-filter architecture (see Fig. 1).

R Parser. An R script named `RParser.R` that the package `xmlparsedata`[14] to parse R files to XML format. Its XML output contains information on line number, column number, start and end positions of each code artifact (e.g., braces, left-assigns, function calls, definitions).

This format is essential for calculating the metrics since a parsed XML tree gives the *RScanner* (the next component) several advantages: (a) multiple metrics can be calculated at once without reparsing the data, (b) enabling a more efficient process, and (c) allowing querying a single subtree node without traversing the entirety of the node. This is relevant, given the existence of multiple OO frameworks [36], and the prevalence of `tidyverse` notation [43], making the static code analysis not be as straightforward as in other languages.

R Scanner. A Python script named `RScanner.py` which receives the *parser*'s XML file as an input. The *Scanner* uses the `xml.etree.ElementTree` package to read the XML file into a manageable, in-memory format, to traverse the tree and extract metrics. Originally, R has no native OO support—R's 'native coding style' is (irrespective of S3 being part of the base distribution) functional programming (see Section 2.1). Therefore, **analyzeR**'s *RScanner* relies on matching patterns to extract metrics.

The *RScanner* outputs a JSON file `metrics.json` (e.g., Listing 1). This file stores metrics calculated for the R package, along with a script version for future debugging and verification. The metrics are presented by acronym and measured value (available Table 1). Overall, metrics are calculated:

- Per File. Calculated for each package file; e.g., in Listing 1 there are two files, `setcomplement.R`, and `asFuzzySet.R`, each with the corresponding metrics and values.
- Per Class. Calculated by class and not by file, regardless of how many files contain the definition of that particular class. The class name is extracted from the corresponding setter method or definition. The metrics calculated by class are: CA, CE, MI, CBO, LCOM (see Table 1).

```
1 {"filename": "./R/operation_setcomplement.R",
2   "LOC": 151, "NMC": 22, "NMCI": 3, "NMCE": 19, "NOF": 0,
3   "NPM": 0, "NPRIF": 0, "NPRIM": 0, "DAM": 0, "AMC": 0,
4   "WMC": 0, "RFC": 0}
5
6 {"filename": "./R/asFuzzySet.R",
7   "LOC": 73, "NMC": 3, "NMCI": 0, "NMCE": 3, "NOF": 0,
```

8 https://contributions.bioconductor.org.

9 https://carpentries-incubator.github.io/bioc-project/05-s4/index.html.

10 Documentation: https://docs.sonarqube.org/8.1/setup/install-plugin/.

11 https://docs.sonarqube.org/latest/analysis/scan/sonarscanner/.

12 TD calculated here is generally based only on code smells.

13 https://docs.sonarqube.org/latest/user-guide/quality-gates/.

14 https://github.com/r-lib/xmlparsedata#readme.

```
8   "NPM": 0, "NPRIF": 0, "NPRIM": 0, "DAM": 0,
9   "AMC": 0, "WMC": 0, "RFC": 0}
10
11 {"className": "UniversalSet", "CA": 0, "CE": 0, "MI": 0, "CBO":
     ↪  1, "LCOM": 1.1111111111111112}
```

Listing 1: Snippet from the metrics.json file.

SonarQube Plugin. The plugin's Java-side uses a custom `SonarSensor` [5], to read the metrics outputted by the *RScanner*. Each metric is implemented using SonarQube's `measures.Metric` API,[15] which defines metric individually, including name, description, and best value (i.e., known as 'grades' or 'indicators'). This includes a 'domain' to categorize the metrics into the groups of Size, Complexity, Couple, Cohesion, and Encapsulation. The first two categories match those traditionally available in SonarQube (see Section 2.2), while the remaining should be interpreted as Maintenance.

2.4. Software functionalities

Currently, **analyzeR** does not compute any R-specific metric, as we prioritized implementing a plugin compatible with current tools that could be easily extended. The measurements are presented through the SonarQube interface to improve readability and comparison with output for other languages. Overall, **analizeR** offers the following functionalities:

- Enables automated static code analyses of OO-structured R packages, and can be incorporated into a CI setup, following current industry practices [4].
- Calculates metrics per-file, per-module, per-class, and per-project. In OO R programming, projects are represented as packages.
- Capabilities to analyze R packages written with S3, S4 or R6, extensible to support additional OO systems, like `proto` and `R.oo`, or newer systems that may become available in the future.
- The *Scanner* is written in Python and favors extensibility to add additional metrics.
- Measurement results are available through the SonarQube interface, respecting the placement and structure provided in other languages' plugins. This allows distinguishing between comparable and custom measurements, aggregation, and results visualization.

2.4.1. Available metrics

analyzeR calculates several metric-types. Size metrics are obtained by counting occurrences in the module under analysis. Complexity metrics are calculated from a package's source files by using the existent `cyclocomp` R package[16]; in general, a low complexity value is seen as desirable.

Essential metrics for OO relate to Coupling and Cohesion. Coupling relates to the number of inter-module connections in a software system, which provides an established way to measure internal software quality concerning modularity [46]. The implemented Coupling metrics indicate loose coupling when the obtained values are low and a tight coupling when high [44]. **analyzeR** includes Afferent and Efferent Coupling, and Coupling Between Objects Classes, derived from Wiese et al. [44]. We also included *Martin's Instability Measure* (MI) [45], where high MI values indicate a high risk of code changes affecting the system's behavior due to other changes in the system.

Cohesion also contributes to modularity since it helps create software components reusable because of their lack of dependence on other components [47]. The only implemented metric allows determining how coupled different function pairs are—thus, lower values are preferred.

Encapsulation refers to hiding internal details (e.g., attributes, internal implementation) to keep unrelated concerns isolated [46], and the application of encapsulation in scientific software has a direct effect on its performance [48]. Here, the number of private (or 'internal', as called in R) fields and methods are calculated by counting their occurrences in a module. Thus, data access metric represents the ratio of the number of private fields and the total number of fields in the module Wiese et al. [44].

Table 1 summarizes all included metrics by name and definition; unless specified, they were implemented as defined by Wiese et al. [44]. Definitions are maintained as-is, but for R packages, an 'application' should be interpreted as a module and/or package.

Two metrics from Wiese et al. [44] were not included in this plugin. The number of Static Fields was not implemented as there are no static fields in R's OO packages, and Cohesion Among Methods since it requires the explicit type of each parameter. Since R is dynamically-typed, there are no reserved words for types, and all method arguments can take upon any value.

2.4.2. Measures visualization

Measurements are usually compared with grades based on the risk they represent. Grades are *indicators* of software quality—a threshold determining how a measure should be interpreted (e.g., 'how much is risky') [49]. Therefore, indicators are essential to static analysis, allowing to understand code quality based on the obtained values [50].

However, given the scarcity of research on R metrics, most grades are generally based on pre-established OO indicators, non-existent for R; therefore, **analyzeR**'s grade-distribution plot remains future work. In SonarQube's interface, file-specific and project-wide metrics are seen as plain values, presented under separate categories in a menu. The interface outlines the name and obtained value for the active file for each of them.

The decision not to show the indicators was taken to: (a) avoid confusion with pre-existing default SonarQube metrics and indicate that they are custom calculated for R, (b) avoid forcing an interpretation that may be incorrect as there is little supporting research, and (c) provide values to enable comparison between languages (enabling future research to determine the indicators).

[15] https://next.sonarqube.com/sonarqube/web_api/api/measures.

[16] https://cran.r-project.org/web/packages/cyclocomp/index.html.

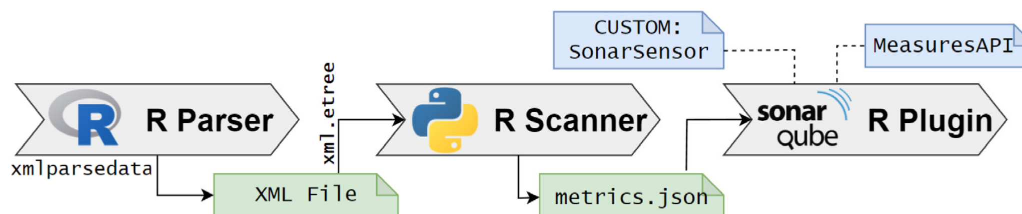

Fig. 1. AnalyzeR's pipe-and-filter architecture components. Green indicates outputs, and blue resources.

Table 1
AnalyzeR available metrics by type. Unless specified, the metrics are from [44].

Acron.	Metric	Description
		Type: Size (File)
LOC	Lines of code	Number of non-blank lines
NPM	# of Public Methods	Number of public functions in an application
NOF	# of Fields	Number of public fields in an application
NMC	# of Method Calls	Number of function invocations
NMCI	# of Internal Method Calls	Number of function invocations of function defined in the application
NMCE	# of External Method Calls	Number of function invocations of function defined in other modules or packages
		Type: Complexity (by File)
WMC	Weighted Methods per Class	Sum of the CycloComp of all functions in the application
AMC	Average Method Complexity	Average of the CycloComp of all functions in the application
RFC	Response For a Class	Number of functions that respond to the application itself
		Type: Coupling (by Class)
CBO	Between Object Classes	How many other modules an application (or module) is coupled to
CA	Afferent Coupling	How many other applications use the specific app./module
CE	Efferent Coupling	How many other modules are used by the specific app./module
MI	Martin's Instability	Indicates the necessity of performing modifications in an entity due to updates in other software entities. [45]
		Type: Cohesion (by Class)
LCOM	Lack of Cohesion in Methods	Difference between the number of function pairs without and with common non static fields
		Type: Encapsulation (by File)
DAM	Data Access Metrics	Ratio of the number of private fields to total number of fields
NPRIF	Number of Private Fields	Number of private fields of an application or module
NPRIM	Number of Private Methods	Number of private functions of an application or module

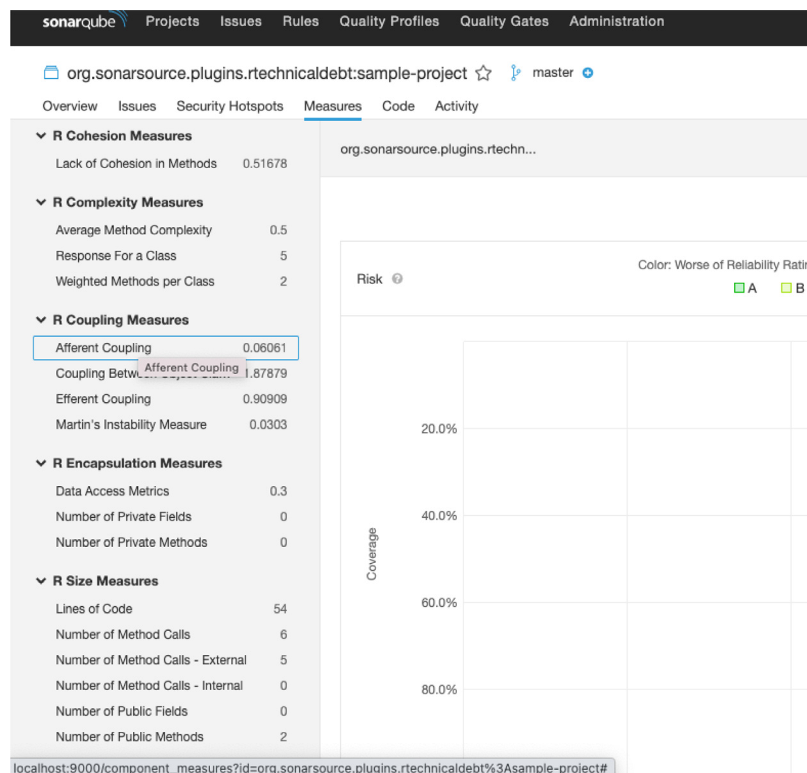

Fig. 2. Measures tab for an example, showcasing the obtained values.

Fig. 2 shows the 'measure tab' for a given file; the list of measures (with highlighted Afferent Coupling) is on the left-hand side. A complete example is available in a video in Section 3.

2.4.3. Adding new metrics

analyzeR was designed for extensibility, and new metrics can be added to the *RScanner*. To do this, a metric's pattern match has to be implemented in the *RScanner* for the OO implementation's constructs and syntax, since they would appear in the parsed tree (namely, the XML file). These metrics must be grouped into file-specific, project-based metrics, or class-specific metrics.

The new metric's calculation must be defined in `RScanner.py`. There are two different inclusion processes, depending on whether they are *derived metrics* or *bespoke metrics*:

(1) **Derived Metrics** use existing metrics and can be implemented using current function calls. For example, Martin's Instability Measure (MI) [45], uses afferent (*CA*) and

efferent (CE) coupling: $MI = CE/(CE + CA)$. Therefore, to calculate MI, the plugin first needs to complete the measurement of CE and CA to use those values in the MI equation.
If the metrics needed are already available, the new metric has to be defined in `RMetrics.java` using the `Metric.Builder` API, and instantiated in `RFileMetric.java`, `RClassMetric.java` or `RModule Metric.java` depending on the desired scope of the metric. If the new metric depends on other metrics not yet available in **analyzeR**, those must be added first as *bespoke metrics*. Non-derived metrics must always be included first.

(2) **Bespoke Metrics.** These metrics do not depend on others, and a new pattern has to be defined in the *RScanner*. This can be achieved by using the `xml.ElementTree` API to determine which node of the *Parser*'s XML file should be targeted and assessed. Once the metric's calculation has been defined, its availability needs to be listed into `metrics.json`. The `metrics` array will then be exported as a `.json` file, used by SonarQube to display results visually. Then, the metric must be defined in `RMetrics.java` as done for a regular *derived metric*.

Finally, the metrics must be written into the output file (namely, `metrics.json`).

Listing 2 presents an example of a function to return a list of function calls. This is later differentiated into internal and external function calls to calculate *NMCE* and *NMCI*.

```
1 def getFunctionCalls(root):
2    """
3    Gets a List of function calls
4    :param root: Root node of an XML Element tree
5    :type root: XML ElementTree root node
6    :return: Function Calls
7    :rtype: List
8    """
9    result = []
10   for c in root.findall('.//SYMBOL_FUNCTION_CALL'):
11      if (c.text in builtins) or (c.text in r6_keywords) or (
   ↪    c.text in s4_keywords):
12         pass
13      else:
14         result.append(c.text)
15   return (list(set(result)))
```

Listing 2: Example function from `RScanner.py`

To have project-wide values calculated for a metric, a new `MeasureComputer` class must be implemented in `*.measures.cumulative`, on the Java side of the plugin. This can implement the aggregate function of choice by providing the corresponding behavior to those available in the interface, measuring project-wide metrics.

As mentioned in Section 2.4.2, for now, the metrics only display raw output (e.g., values without indicators) because the grades are not available; hence, the visual plots remain empty. However, once the indicators become available in the future, it will be possible to associate 'grades' and 'best values' with them, enabling the plots. To add grades and best values for a metric, they must be set in the metric's `Metric.Builder` in `RMetrics.java`.

Currently, display/hiding metrics based on the OO system (namely, to select which metrics to calculate for each OO implementation) has not been implemented. For example, a user cannot choose to skip MI when examining a package built on S3. Therefore, *non-applicable metrics* are displayed alongside applicable metrics but showcased values of a dash. New OO-specific metrics can be added but will be considered non-applicable for the other OO implementations. For example, a user-extension can add metric X for R6 only, and if an S3/S4 package is examined, metric X will be shown as $X = 0$ in the results. Since all three systems use `.R` files, it is not possible to have different metrics for each file. Moreover, packages can use combinations of the OO implementations, and such detection would require further code analyses, deemed as low-priority requirements for **analyzeR**'s first release.

If there is a new category of metrics beyond those currently available, **analyzeR**'s Java-side must be extended following the same structure as `RProjectMetrics. java`, `RFileMetrics. java`, `RModuleMetrics.java` and `RClassMetrics.java`, which are the classes implementing the logic to categorize metrics in the available categories.

3. Illustrative examples

This example is based on analyzing `set6`[17] package, chosen because it is up-to-date and contains multiple, well-implemented R files that can thoroughly display the **analyzeR**'s capabilities. Although `set6` is implemented with R6, no additional steps are needed to analyze S3 or S4-coded packages, as the plugin automatically detects it. While the demonstration below highlights necessary steps, detailed instructions are included with the documentation. Additionally, an explanatory video is provided alongside this article.

Installing the plugin

SonarQube must be installed on a computer to use **analyzeR**. The steps are available on SonarQube's documentation and are external to our plugin: https://docs.sonarqube.org/latest/setup/overview/.

analyzeR is provided as a `*.jar` file as requested by SonarSource. Although multiple releases may be available, it is recommended to check those compatible with the installed SonarQube version. Once the `.jar` file has been downloaded, it must be copied into `$SONARQUBE_HOME/extensions/plugins`; any prior versions **analyzeR** must be removed from that folder before restarting SonarQube.

Then, the SonarQube server must be started, depending on how it was installed. The same link for the official documentation provides different alternatives.

Analyzing an R package

Packages cannot be analyzed online and must be downloaded to the server. In R package's root directory (where the `*.Rproj` file is located), a new file called `sonar-project.properties` must be created, following the structure presented in Listing 3. This file must contain line #10 (as per the Listing), to ensure the *Scanner* component outputs the corresponding `metrics.json` file.

```
1 # SonarQube Project Properties.
2 sonar.projectKey=orq.sonarsource.plugins.rtechnicaldebt:
     ↪ set6-demo
3 sonar.projectName=set6-demonstration
4 sonar.sources=R
5 sonar.host.url=http://localhost:9000
6 sonar.sourceEncoding=UTF-8
7
8 # This is the metrics file. Do not change this.
9 sonar.r.tdebt.output=metrics.json
```

Listing 3: Example `sonar-project.properties` file.

[17] https://github.com/xoopR/set6.

Fig. 3. SonarQube after successfully running the SonarScanner (Excerpt).

Then, **analyzeR** can be used. The analyzing process is initiated through a console command requiring to: (1) navigate towards the R folder of the project (in this example, `set6/R`), and (2) initiate the scan with `python3 RScanner.py ./R/*.R`.

Once activated, **analyzeR** will explore the R package, listing the processed files as console output. This process cannot be interrupted, or the results will be corrupted, and the scan will have to be restarted. The time needed to complete the process depends on the number of files to scan and measure and the number of implemented metrics.

To continue, the user must ensure they have a working login ID and password to the corresponding SonarQube instance, or SonarQube itself will not be accessible. When running SonarQube over Windows, the path to the SonarScanner installation must have been previously added to the environmental variable `$PATH` (for the entire computer, and not only for the user). If this is not done, the following commands will fail due to SonarQube's errors, extraneous to **analyzeR**. To understand which variables must be set, we refer developers to SonarQube's official installation manual.[18]

Once the `metrics.json` has been created for that particular package, and the above data steps are completed, the *RPlugin* can be run to obtain a visualization of the metrics. This is done with another console command, executed in the packages' R folder: `sonar-scanner -Dsonar.login=<login-id> -Dsonar.password=<sonar-password>` (replace with the corresponding *login ID* and *password*). After the plugin has processed all corresponding files, the results become visible in SonarQube's 'Code' tab (see Fig. 3).

Project-wide metrics are displayed under the 'Measures Tab' to differentiate them from the default SonarQube measures, exhibited separately (Fig. 4). Limitations and availability were explained in Section 2.4.2, and will be obtained in a similar style as seen in Fig. 2.

To view detailed measures for each file, the developer must browse the scanned files in the interface (Fig. 3), and click the corresponding file name. This will load an interface to explore that file's source-code. On the top-right corner, there is a three-line mobile menu (indicated with a red arrow in Fig. 5). After clicking on it, the option to select is 'Show All Measures'.

These can also be accessed from the 'Measures' tab after navigating to the corresponding file. Class-specific metrics are displayed only within the terminal (both for Linux/Windows) without needing an additional command (see Listing 4). Each class is stated in an individual line by name, followed by the metrics' acronym and the value measured.

```
1 Class Class = Properties{CBO=1, CA=0, CE=0, MI=0.0, LCOM=1.0434782608695652}
2 Class Class = Set{CBO=11, CA=0, CE=0, MI=0.0, LCOM=1.0153846153846153}
3 Class Class = SetWrapper{CBO=4, CA=0, CE=0, MI=0.0, LCOM=1.0909090909090908}
4 Class Class = ComplementSet{CBO=1, CA=0, CE=0, MI=0.0, LCOM=1.05}
5 Class Class = ExponentSet{CBO=1, CA=0, CE=0, MI=0.0, LCOM=1.0909090909090908}
6 Class Class = ProductSet{CBO=1, CA=0, CE=0, MI=0.0, LCOM=0.0}
7 Class Class = PowersetSet{CBO=1, CA=0, CE=0, MI=0.0, LCOM=0.0}
8 Class Class = UnionSet{CBO=1, CA=0, CE=0, MI=0.0, LCOM=0.0}
9 Class Class = Complex{CBO=1, CA=0, CE=0, MI=0.0, LCOM=0.0}
10 Class Class = ConditionalSet{CBO=2, CA=0, CE=0, MI=1.0, LCOM=1.0384615384615385}
```

Listing 4: Example of Class Metrics displayed within the Terminal.

Fig. 4. Metrics displayed on the left side of the 'Measures' tab in SonarQube.

4. Impact

ASATs impact software development lifecycles and code quality analyses [1,2]. They are also shaping junior developers to think about quality while implementing as companies continue to focus on TD management [4,23,30]. However, most TD research

[18] https://docs.sonarqube.org/latest/setup/overview/.

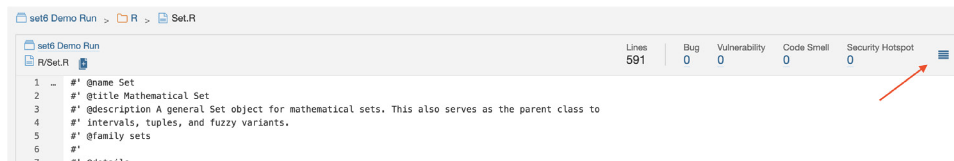

Fig. 5. Location of the menu to find file-specific metrics while inspecting a file's source-code.

has traditionally focused on OO, statically-typed software, specially coded in Java [6,22,24]. Only recent efforts moved research towards other languages [15,16,18,25,26].

Regardless of its increasing popularity, R's static analysis remains behind similarly widespread languages. Its multi-paradigm, package-based, and special-purpose characteristics make ASATs of utmost importance to support not only the development of new packages but also the quality of the research based on them [11,13,14,17].

analyzeR will also assist **existing user communities**:

- Specialized R Package peer-review organizations (e.g., BioConductor and rOpenSci) currently use `goodpractices` as their primary ASAT, but rely on a completely manual analysis [18]. Although code peer-review cannot be replaced by an automated approach, **analyzeR** could considerably improve the process by easing code examination. It is known that although R has OO features, the community generally regards them as 'advanced' [14,36,37], and thus are not as widespread as the traditional functional approach.
- *Research Software Engineers* (RSE) are software developers working on scientific software, thus having an intrinsic difference to their industry counterparts [12,13]. "Research software engineers believe that an increased focus on quality and discoverability are key factors in increasing the sustainability of academic research software" [51], which differs from the security, bug-free centered approach of 'industry' software developers. Moreover, their increased challenges have given rise to multiple RSE organizations, deemed "crucial to the long-term health of research software" [52]. Therefore, **analyzeR** can provide support for software sustainability by blending technical software engineering knowledge and state-of-the-art approaches, eased into automated analysis through a tool that already provides continuous integration support. It is straightforward to use and set up, enabling its use in the classroom as well.
- Given that (a) **analyzeR** is open-source, (b) there is thorough documentation available, and (c) it was built to be extensive, any R developer with relevant knowledge could contribute to the plugin, ensuring it remains up-to-date.

Therefore, **analyzeR** enables many lines for **future research works**:

- SonarQube does not allow class-based metrics to be displayed, only supporting file-level and project-level metrics. However, we can calculate class-based metrics in the *RScanner*. If SonarQube allows supporting class-level metrics in the future, these can also be added. Some are already present in the current `metrics.json`, only needing to be displayed within SonarQube's visualizations.
- Although multiple investigations leveraged ASATs to study TD in Java or Python, R Programming has not been addressed as a domain due to limited existing tools (see Section 1.1). Therefore, analyzeR will enable *new TD research* for R packages and allow *comparing findings* with other languages to further understand the differences between domains.
- Most of the static metrics provided exclusively for R do not include indicators (e.g., guidelines on how to interpret the measures) [44]. Likewise, those imported from other languages adopt foreign indicators without assessing their correctness (e.g., test coverage [16]). Through **analyzeR**, it will be possible to mine software repositories and explore code on a large scale to obtain these indicators.
- **analyzeR** enables large-scale mining software repositories research that otherwise would not be possible. For example, to investigate the evolution of TD and code quality in R Packages of different characteristics or compare those built on top of either OO implementation.
- An essential element of **analyzeR**'s features and design was the ability to compare results to those obtained with SonarQube plugins for other languages (e.g., Python, Java, among others). This comparability enables future assessments on the differences between R Programming and other paradigms and domains, benefiting R and similar programming languages.

SonarQube offers support for almost 25 programming languages but mainly supports traditional, general-purpose programming languages [6]. Moreover, those languages can also be used as special-purpose (e.g., Python) have limited support in this regard. Although **analyzeR** cannot be extended to other programming languages, the metrics implementations, the code's scan, and the architecture in itself can be generalized to similar languages beyond the intended user group (namely, R developers or RSEs working with R Programming).

5. Conclusions

analyzeR is an extensible, flexible plugin to peruse object-oriented R Packages through SonarQube. It supports all three of R's object-oriented implementations (S3, S4, R6), including visual support and results comparable to other languages. Its implementation combines R, Python and Java, and the resulting tool can be incorporated into most continuous-integration environments. **analyzeR** is an ongoing project. We intend to improve the metrics and incorporate grades (i.e., metrics indicators) once the theory behind them has been completed. Additionally, we plan to extend the features to incorporate graphs, visualizations, and other metrics not added in the first iteration.

Declaration of competing interest

The authors declare the following financial interests/personal relationships which may be considered as potential competing interests: Dr Zadia Codabux reports financial support was provided by Natural Sciences and Engineering Research Council of Canada.

Acknowledgments

This study is partly supported by the Natural Sciences and Engineering Research Council of Canada, RGPIN-2021-04232 and DGECR-2021-00283 at the University of Saskatchewan.

Appendix A. Supplementary data

Supplementary material related to this article can be found online at https://doi.org/10.1016/j.softx.2022.101113.

References

[1] Marcilio D, Bonifácio R, Monteiro E, Canedo E, Luz W, Pinto G. Are static analysis violations really fixed? A closer look at realistic usage of SonarQube. In: IEEE/ACM 27th International conference on program comprehension. 2019, p. 209–19. http://dx.doi.org/10.1109/ICPC.2019.00040.

[2] Vassallo C, Panichella S, Palomba F, Proksch S, Zaidman A, Gall HC. Context is king: The developer perspective on the usage of static analysis tools. In: International conference on software analysis, evolution and reengineering. 2018, p. 38–49. http://dx.doi.org/10.1109/SANER.2018.8330195.

[3] Vassallo C, Palomba F, Bacchelli A, Gall HC. Continuous code quality: Are we (Really) doing that? In: 33rd ACM/IEEE International conference on automated software engineering. New York, NY, USA: Association for Computing Machinery; 2018, p. 790–5.

[4] Lenarduzzi V, Lomio F, Huttunen H, Taibi D. Are SonarQube rules inducing bugs? In: International conference on software analysis, evolution and reengineering. 2020, p. 501–11. http://dx.doi.org/10.1109/SANER48275.2020.9054821.

[5] SonarSource SA, Switzerland. SonarQube Documentation. 2021.

[6] Saarimaki N, Baldassarre MT, Lenarduzzi V, Romano S. On the accuracy of SonarQube technical debt remediation time. In: Euromicro conference on software engineering and advanced applications. 2019, p. 317–24. http://dx.doi.org/10.1109/SEAA.2019.00055.

[7] Nielsen ME, Østergaard Madsen C, Lungu MF. Technical debt management: A systematic literature review and research agenda for digital government. In: Viale Pereira G, Janssen M, Lee H, Lindgren I, Rodríguez Bolívar Manuel P, Scholl HJ, et al., editors. Electronic government. Cham: Springer International Publishing; 2020, p. 121–37.

[8] Maldonado ES, Shihab E. Detecting and quantifying different types of self-admitted technical debt. In: 7th International workshop on managing technical debt. Bremen, Germany: IEEE; 2015, p. 9–15. http://dx.doi.org/10.1109/MTD.2015.7332619.

[9] da Silva Maldonado E, Shihab E, Tsantalis N. Using natural language processing to automatically detect self-admitted technical debt. IEEE Trans Softw Eng 2017;43(11):1044–62. http://dx.doi.org/10.1109/TSE.2017.2654244.

[10] Baek N, Kim Kuinam J. Prototype implementation of the OpenGL ES 2.0 shading language offline compiler. Cluster Comput 2019;22(1):943–8. http://dx.doi.org/10.1007/s10586-017-1113-z.

[11] Howison J, Deelman E, McLennan MJ, Ferreira da Silva R, Herbsleb JD. Understanding the scientific software ecosystem and its impact: Current and future measures. Res Eval 2015;24(4):454–70. http://dx.doi.org/10.1093/reseval/rvv014.

[12] Hannay JE, MacLeod C, Singer J, Langtangen HP, Pfahl D, Wilson G. How do scientists develop and use scientific software? In: ICSE Workshop on software engineering for computational science and engineering. Vancouver, Canada: IEEE; 2009, p. 1–8. http://dx.doi.org/10.1109/SECSE.2009.5069155.

[13] Arvanitou E-M, Ampatzoglou A, Chatzigeorgiou A, Carver JC. Software engineering practices for scientific software development: A systematic mapping study. J Syst Softw 2021;172:110848. http://dx.doi.org/10.1016/j.jss.2020.110848.

[14] Pinto G, Wiese I, Dias LF. How do scientists develop scientific software? An external replication. In: 25th International conference on software analysis, evolution and reengineering. Campobasso, Italy: IEEE; 2018, p. 582–91. http://dx.doi.org/10.1109/SANER.2018.8330263.

[15] Vidoni M. Self-admitted technical debt in R packages: An exploratory study. In: International conference on mining software repositories. Madrid, Spain: IEEE; 2021, p. 179–89.

[16] Vidoni M. Evaluating unit testing practices in R packages. In: 43rd International conference on software engineering. Madrid, Spain: IEEE; 2021, p. 1–12.

[17] Storer T. Bridging the chasm: A survey of software engineering practice in scientific programming. ACM Comput Surv 2017;50(4). http://dx.doi.org/10.1145/3084225.

[18] Codabux Z, Vidoni M, Fard FH. Technical debt in the peer-review documentation of r packages: A rOpenSci case study. In: IEEE/ACM 18th International conference on mining software repositories. USA: IEEE; 2021, p. 195–206. http://dx.doi.org/10.1109/MSR52588.2021.00032.

[19] Vidoni M. Software engineering and R programming: A call for research. R J 2021;13(2):600–23. http://dx.doi.org/10.32614/RJ-2021-108.

[20] Vidoni M. Understanding roxygen package documentation in R. J Syst Softw 2022;188:111265. http://dx.doi.org/10.1016/j.jss.2022.111265.

[21] Khan JY, Uddin G. Automatic detection and analysis of technical debts in peer-review documentation of r packages. 2022, CoRR, abs/2201.04241 arXiv:2201.04241.

[22] Baldassarre MT, Lenarduzzi V, Romano S, Saarimäki N. On the diffuseness of technical debt items and accuracy of remediation time when using SonarQube. Inf Softw Technol 2020;128:106377. http://dx.doi.org/10.1016/j.infsof.2020.106377.

[23] Avgeriou PC, Taibi D, Ampatzoglou A, Arcelli Fontana F, Besker T, Chatzigeorgiou A, et al. An overview and comparison of technical debt measurement tools. IEEE Softw 2021;38(3):61–71. http://dx.doi.org/10.1109/MS.2020.3024958.

[24] Lenarduzzi V, Saarimäki N, Taibi D. The technical debt dataset. In: International conference on predictive models and data analytics in software engineering. PROMISE'19, New York, NY, USA: Association for Computing Machinery; 2019, p. 2–11. http://dx.doi.org/10.1145/3345629.3345630.

[25] Tan J, Feitosa D, Avgeriou P, Lungu M. Evolution of technical debt remediation in Python: A case study on the apache software ecosystem. J Softw Evol Process 2021;33(4):e2319. http://dx.doi.org/10.1002/smr.2319, e2319 smr.2319.

[26] Tan J, Feitosa D, Avgeriou P. An empirical study on self-fixed technical debt. In: 3rd International conference on technical debt. TechDebt '20, New York, NY, USA: Association for Computing Machinery; 2020, p. 11–20. http://dx.doi.org/10.1145/3387906.3388621.

[27] Tan J, Feitosa D, Avgeriou P. Investigating the relationship between co-occurring technical debt in Python. In: Euromicro conference on software engineering and advanced applications. 2020, p. 487–94. http://dx.doi.org/10.1109/SEAA51224.2020.00082.

[28] Lenarduzzi V, Martini A, Taibi D, Tamburri DA. Towards surgically-precise technical debt estimation: Early results and research roadmap. In: Proceedings of the 3rd ACM SIGSOFT International workshop on machine learning techniques for software quality evaluation. MaLTeSQuE 2019, New York, NY, USA: Association for Computing Machinery; 2019, p. 37–42. http://dx.doi.org/10.1145/3340482.3342747.

[29] Tsoukalas Dimitrios, Kehagias Dionysios, Siavvas Miltiadis, Chatzigeorgiou Alexander. Technical debt forecasting: An empirical study on open-source repositories. J Syst Softw 2020;170:110777. http://dx.doi.org/10.1016/j.jss.2020.110777.

[30] Gilson F, Morales-Trujillo M, Mathews M. How junior developers deal with their technical debt? In: International conference on technical debt. TechDebt '20, USA: Association for Computing Machinery; 2020, p. 51–61. http://dx.doi.org/10.1145/3387906.3388624.

[31] Molnar A-J, Motogna S. Long-term evaluation of technical debt in open-source software. In: 14th ACM/IEEE International symposium on empirical software engineering and measurement. ESEM '20, New York, NY, USA: Association for Computing Machinery; 2020, p. 1–9. http://dx.doi.org/10.1145/3382494.3410673.

[32] Capilla R, Mikkonen T, Carrillo C, Fontana FA, Pigazzini I, Lenarduzzi V. Impact of opportunistic reuse practices to technical debt. In: IEEE/ACM International conference on technical debt. 2021, p. 16–25. http://dx.doi.org/10.1109/TechDebt52882.2021.00011.

[33] Digkas G, Nikolaidis N, Ampatzoglou A, Chatzigeorgiou A. Reusing code from StackOverflow: The effect on technical debt. In: 45th Euromicro conference on software engineering and advanced applications. 2019, p. 87–91. http://dx.doi.org/10.1109/SEAA.2019.00022.

[34] Codabux Z, Dutchyn C. Profiling developers through the lens of technical debt. In: International symposium on empirical software engineering and measurement. ESEM '20, New York, NY, USA: Association for Computing Machinery; 2020, p. 1–6. http://dx.doi.org/10.1145/3382494.3422172.

[35] Turcotte A, Vitek J. Towards a type system for r. In: Workshop on implementation, compilation, optimization of object-oriented languages, programs and systems. ICOOOLPS '19, London, United Kingdom: Association for Computing Machinery; 2019, p. 1–5. http://dx.doi.org/10.1145/3340670.3342426.

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

[37] German DM, Adams B, Hassan AE. The evolution of the r software ecosystem. In: European conference on software maintenance and reengineering. Genova, Italy: IEEE; 2013, p. 243–52. http://dx.doi.org/10.1109/CSMR.2013.33, ISSN: 1534-5351.

[38] Chambers JM. Object-oriented programming, functional programming and R. Statist Sci 2014;29(2):167–80. http://dx.doi.org/10.1214/13-STS452.

[39] Adler J. R in a nutshell. O'Reilly Media, Inc.; 2012.

[40] Chang Winston. R6: Encapsulated Classes with Reference Semantics. 2021.

[41] Bengtsson H. The r. oo package-object-oriented programming with references using standard R code. In: 3rd International workshop on distributed statistical computing. Vienna, Austria: Austrian Association for Statistical Computing (AASC); 2003, p. 20–2.

[42] Kates L, Petzoldt T. proto: An R Package for Prototype Programming. Technical report, Technische Universit.
[43] Wickham H, Grolemund G. R for Data Science: Import, Tidy, Transform, Visualize, and Model Data. 1st. O'Reilly Media, Inc.; 2017.
[44] Wiese L, Wiese I, Lietz K. Software quality assessment of a web application for biomedical data analysis. In: 25th International database engineering and applications symposium. IDEAS 2021, New York, NY, USA: Association for Computing Machinery; 2021, p. 84–93. http://dx.doi.org/10.1145/3472163.3472172.
[45] Santos DB, Resende AMP, Lima EC, Freire AP. Software instability analysis based on afferent and efferent coupling measures. J Softw 2017;12(1):19–34. http://dx.doi.org/10.17706/jsw.12.1.19-34.
[46] Schnoor H, Hasselbring W. Comparing static and dynamic weighted software coupling metrics. Computers 2020;9(2). http://dx.doi.org/10.3390/computers9020024.
[47] Rathee A, Chhabra JK. Improving cohesion of a software system by performing usage pattern based clustering. Procedia Comput Sci 2018;125:740–6. http://dx.doi.org/10.1016/j.procs.2017.12.095.
[48] Pizarro-Vasquez GO, Barahona F, Botto-Tobar M. Encapsulation component and its incidence into scientific software performance. In: Rocha Á, López-López PC, Salgado-Guerrero JP, editors. Communication, smart technologies and innovation for society. Singapore: Springer Singapore; 2022, p. 709–19.
[49] Medeiros N, Ivaki N, Costa P, Vieira M. Vulnerable code detection using software metrics and machine learning. IEEE Access 2020;8:219174–98. http://dx.doi.org/10.1109/ACCESS.2020.3041181.
[50] Dey T, Mockus A. Deriving a usage-independent software quality metric. Empir Softw Eng 2020;25(2):1596–641. http://dx.doi.org/10.1007/s10664-019-09791-w.
[51] Rosado de Souza M, Haines R, Vigo M, Jay C. What makes research software sustainable? An interview study with research software engineers. In: International workshop on cooperative and human aspects of software engineering. 2019, p. 135–8. http://dx.doi.org/10.1109/CHASE.2019.00039.
[52] Carver JC, Cosden IA, Hill C, Gesing S, Katz DS. Sustaining research software via research software engineers and professional associations. In: IEEE/ACM International workshop on body of knowledge for software sustainability. 2021, p. 23–4. http://dx.doi.org/10.1109/BoKSS52540.2021.00016.