Evaluating Unit Testing Practices in R Packages

Melina Vidoni
RMIT University, School of Computing Technologies
Melbourne, Australia
melina.vidoni@rmit.edu.au

***Abstract*—Testing Technical Debt (TTD) occurs due to shortcuts (non-optimal decisions) taken about testing; it is the test dimension of technical debt. R is a package-based programming ecosystem that provides an easy way to install third-party code, datasets, tests, documentation and examples. This structure makes it especially vulnerable to TTD because errors present in a package can transitively affect all packages and scripts that depend on it. Thus, TTD can effectively become a threat to the validity of all analysis written in R that rely on potentially faulty code. This two-part study provides the first analysis in this area. First, 177 systematically-selected, open-source R packages were mined and analysed to address quality of testing, testing goals, and identify potential TTD sources. Second, a survey addressed how R package developers perceive testing and face its challenges (response rate of 19.4%). Results show that testing in R packages is of low quality; the most common smells are inadequate and obscure unit testing, improper asserts, inexperienced testers and improper test design. Furthermore, skilled R developers still face challenges such as time constraints, emphasis on development rather than testing, poor tool documentation and a steep learning curve.**

I. Introduction

There are many popular languages, tools and environments for statistical computing. R (and its accompanying software environment) is a robust open-source competitor, regardless of how popularity is measured [1, 2]. R is a package-based programming ecosystem and provides an easy way to install third-party code, datasets, tests, documentation and examples [3]. The main R distribution installs a few base and recommended packages. Though CRAN (Comprehensive R Archive Network)[1] distributes R packages, many are developed on public version control repositories.

Technical Debt (TD) is a metaphor used to describe the situation where long-term software code quality is traded for a quick, short-term solution —it incurs in future costs of maintenance, adaptability and revisions [4]. It is a metaphor used to reflect the implied cost of additional rework caused by choosing an easy solution in the present, instead of using a better approach that would take longer [4]. *Testing Technical Debt* (TTD) occurs due to shortcuts (non-optimal decisions) related to testing, and it is the test dimension of TD [5]. This is because not performing unit testing or writing incomplete tests can speed up development, at the cost of lowering the quality of the software and increasing the number of faults.

Addressing the problem of inadequate software testing requires defining what this means. Since R is a package-based programming language, errors present in one can transitively affect all packages and scripts that depend on it, effectively becoming a threat to their validity of the analysis written after that. Therefore, it is essential to thoroughly test R packages before releasing them. Common R packages for unit testing R code rely on concepts developed for object-oriented (OO) programming [6]. However, R is a dynamic language that combines lazy functional features with minimal OO features [2]. This combination of paradigms can set it apart from the current OO-focused knowledge of TTD.

Although this problem has been explored in other domains [7, 8, 9], it has not been addressed in R so far. Despite the growth of the R community, there has been (to the author's knowledge) no research assessing how developers test these packages in practice. This study is needed to understand how unit testing is approached, what are the TTD"hot-spots" in R packages and what challenges developers face.

This manuscript addresses this need through a two-part empirical study. The first part analysed 177 systematically-selected, open-source R packages hosted on GitHub; the dataset included both popular and newer packages, maintained since 2018. Both automated and manual analysis were conducted to address quality of testing, to define what is being tested, and to identify harmful practices that increase TTD in R packages. The second part of the study addressed how R developers perceive unit testing; it involved contacting the developers of the packages selected before to ask them to complete an anonymous survey. The survey was sent to 469 developers and received 91 responses (response rate of 19.4%).

The main contributions of this study are as follows:

- This is the first study conducted to understand unit testing culture in the R packages development community. It analyses the extent to which R packages are tested by mining open, public R packages repositories hosted in GitHub, by performing a number of analysis.
- About testing practices, findings indicate that few alternative paths are tested, and test suits seem to be irrelevant since bugs are often found when tests are passing. Developers lack in-depth training, expected due to the absence of technical programming background [10]. Moreover, there is a striking amount of manual testing, potentially due to the statistical nature of R packages, and the lack of statistically centred unit testing.
- Regarding existing testing tools, results show that R's unit testing tools may be incomplete, lacking assertion functionalities and automatic data initialisation (e.g. Junit's `@BeforeAll`).

[1] https://cran.r-project.org/

- It surveys many developers to understand their perspective in testing tools and challenges faced by them when testing their R packages.

This paper is structured as follows. Section II explains the study design, including research questions, data collection, and survey design. Results are presented in Section III. After that, Section IV discusses additional interesting points and describes threats to the validity of the previous sections. Finally, Section V concludes the paper and mentions future lines of work.

A. Related Works

Studies related to R programming practices are scarce. Decan et al. [11] explored how the use of GitHub influences the R ecosystem, both for the distribution of R packages and inter-repository package dependency management, while Ramirez et al. [12] studied their maintenance to explore change frequency and variability between versions. Furthermore, Morandat et al. [2] assessed the success of different R features to evaluate the fundamental choices behind the language design.

Furthermore, two-part studies combining of mining software repositories (MSR) and developer surveys are present in different areas of software engineering research. For example, it has been used to assess the popularity of open-source repositories in GitHub (stars and watches) [13], as well as to evaluate the reasoning behind forking repositories [14]. In terms of testing, it was used to analyse code coverage visualisations in closed-source software [15], but also to determine TTD in Scala projects and identify potential testing smells [16]. Other authors used it to assess the quality (in terms of smells) of external contributions in GitHub projects [17].

Finally, Křikava and Vitek [18] did study unit tests in R packages, but with different research questions. They conducted an MSR to inspect R packages source code and propose a tool that automatically generates unit tests. They reported the implementation and empirical evaluation of the proposed tool. However, They did not focus on other topics related to unit testing cultures, such as technical debt constraints or the challenges faced by developers.

II. Study Design

This section presents the design of the study. This includes research questions investigated in this study, how data was collected through an MSR approach, survey design and distribution.

The methodology used in this manuscript, and described in the sections below, was approved by RMIT University Human Ethics Research Committee (HREC), with project code 2020-22968-10378.

A. Research Questions

The goal of this study is to understand how R packages are tested in order to define best testing practices in this language as well as to identify Testing TD hot-spots. This leads to the following research questions (RQs):

RQ1. Are R packages well tested? To understand which testing tools are used in R packages, identify common practices, types of testing, and how unit testing tailors to a multi-paradigm language like R.

RQ2. Which are potential Testing TD weak-spots? To discover and understand negative practices that affect unit testing in R packages. The long-term goal of this is to identify testing smells.

RQ3. How do R package developers perceive unit testing? Part of the MSR involved collecting public email addresses of developers, disclosed in packages' files, to send them a structured survey. Questions aimed to understand their subjective perception of testing and the challenges they face.

B. Mining R Packages (Part I)

The first part of the study addresses RQ1 and RQ2 by mining open-source R package repositories hosted in GitHub. The following inclusion and exclusion criteria were used to search repositories:

- Inclusion Criteria: The repository must be an R-package, originally posted during or after 2010; it needs to show maintenance activity (commits) in the last two years (i.e. from 2018). It must have a correct package structure, with all dependencies available.
- Exclusion Criteria: The repository is an R data package, a book, or a personal package. The state of the repository is archived, deprecated, or outdated. It is an R package with scripts used in a book. It has incomplete or missing files (i.e. description, namespace, or readme files). It is a fork from another R package.

The gathering followed a systematic selection process, summarised in Figure 1. GitHub's Advanced Search was used

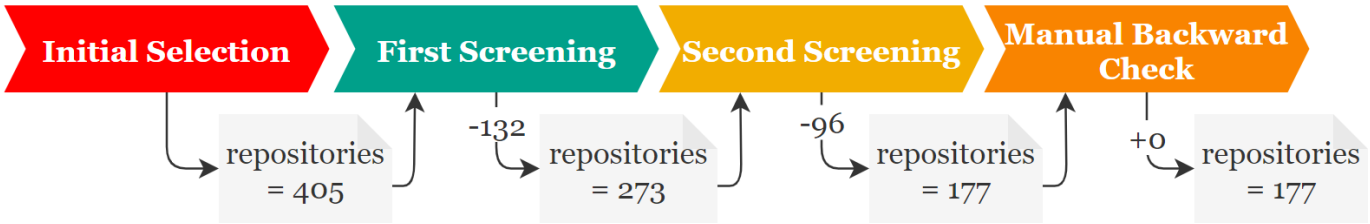

Fig. 1. Repository selection filtering steps.

with the keywords `language:R package` and configured to automatically filter results by the date of the last push, as well as by words (specifically, 'deprecated', 'outdated', 'archived', 'personal') and fork status. This search produced about 27000 packages, and 405 were selected for this process, at the *Initial Selection* step. This number was a threshold defined by considering how many repositories were used on other MSR papers with mostly manual analysis.

For the *first screening*, GitHub metadata was used to exclude repositories not automatically filtered. The folder structure in the main branch was used to determine a correct package structure. Remaining packages were listed in a text file using their slugs (i.e. `username/repositoryName`). This was used in the following phase.

The *second screening* implied automatically downloading the repositories listed in the entry text file, running basic tests to determine suitability according to the remaining aspects of the inclusion/exclusion criteria. This was done using `covr`[2] and `testthat`[3]. For each package, this required downloading the source from GitHub, installing it along with all dependencies, running `covr` to obtain a basic analysis of the tests, and `devtools::test` to run all unit testings. This automated double-checking ensured an analysis of the package's integrity but also their test suitability while discarding possible errors in the cloning.

After this process, **177 packages remained after all selection steps**. Finally, the repositories were further filtered according to the results obtained from this analysis; Table I provides an overview, with the number of packages removed at each stage.

TABLE I
SECOND SCREENING FILTERING RESULTS, WITH `covr` AND `testthat` RESULTS.

`covr`	`testthat`	Result	Number
Yes	Yes	Both analysis run correctly	159
Yes	Failed	`covr` runs, but there are issues with `testthat`	18
NA	NA	Analysis are unable to run. Empty test structure, or manual test cases	45
NA	Manual		20
Error	Yes	Filtered. `covr` is unable to complete the analysis. `testthat` provides mixed results	19
Error	Failed		6
Error	NA		6

In the above table, "failed" means all tests included in the package are unit tests, but most of them failed and produced an early stop; `NA` is an R-specific value that indicates missing results (i.e. tests are not available). Moreover, `covr` has a known bug with dependencies when installing packages in a specific directory[4]. Though in one case packages had a correct unit testing structure (Table I, row 5), it was not possible to analyse them. This will be further discussed when addressing the threats to the validity of this study (see Section IV-D).

Finally, a manual backward check (see Figure 1) was performed to evaluate repositories discarded by the automated checking. This step ensured that both analyses were running correctly. No repositories were added after this.

[2]https://covr.r-lib.org/

[3]https://testthat.r-lib.org/

[4]See: https://github.com/r-lib/covr/issues/248

C. Developers Survey (Part II)

For this part of the study, an anonymous online survey was prepared. Table II details the questions and answers included in it. This survey was implemented in Qualtrics and used its services to distribute it. Email-related information was removed to ensure the anonymity of respondents. Questions labelled with "[S]" indicate a single choice answer.

TABLE II
STRUCTURE OF THE SURVEY GENERATED FOR PART II.

Question	Possible Answers
How many R packages have you authored? (Regardless if they are in CRAN/Bioconductor or not)	<2 packages / 2-5 / 5-10 / >10
How many years of experience do you have as an R programmer?	<2 years / 2-5 / 5-10 / 10+ years
How do you test your code? [S]	Manually / I don't test / Using testing packages
What type of testing do you do?	Individual Functions Only / Functions Clusters / Using my package externally / Other
If you use testing packages, what are the names of them?	Comment box
Why do you use testing packages?	Generating or executing test cases / Creating and evaluating results / Analysing code coverage / Finding bugs / Reporting bugs / Fulfilling CRAN requirements.
Do you face the following challenges during testing? And if you do, how serious are they?	Likert Scale 1-5
What are the top two things you look for/need/would like to see?	Comment box
Do you use coverage visualisation tools? [S]	Always / Occasionally / Never
Name the coverage visualisation tools that you use.	Comment box
Does coverage visualisation affect you? [S]	It motivates me / It makes me anxious / It makes me confident in my code / I trust my code is bug-free / Other
Did you ever have all tests passing, but found a bug in your code? [S]	Yes, at least once / Yes, more than one time / I don't remember / Never

The question regarding challenges included the following options: time constraints, compatibility issues, lack of exposure to tools, emphasis on development rather than testing, lack of support from their organisation, unclear testing benefits, poor documentation, lack of experience and steep learning curve.

Regarding distribution, R packages list information about their maintainers in the 'description' file, including their email address; these emails are publicly shown in CRAN (if the package is available there) as well as in the IDEs (Integrated Development Environment) upon installation. Because of this, an R script was used to extract this information and remove the duplicates, obtaining 469 email addresses. Qualtrics' distribution tools were used to send the survey to potential candidates. Overall, 22 emails bounced, and 91 replies were collected (response rate of 19.4%).

It is worth highlighting that this methodology underwent a thorough ethical review and was approved (see Section II). To preserve the identity of those contacted for the survey, as well as avoiding a potential re-identification of the responses, the names of the packages explored in this manuscript (see Section II-B) are not shared. As a result, the datasets used are not made public.

III. Results

This section reports the result of analysing the testing and coverage of 177 systematically-selected, open-source R packages, as well as the 91 survey responses. Results are presented by RQ to be addressed.

It is worth noticing that during the filtering phase, several repositories were excluded due to testing quality (see Table I). First, 20 repositories were excluded because they had manual testing only. This implied having correct package structure and testing folders (using `testthat` folder hierarchy), but without including asserts in test methods. Thus, the "tests" were manual, and packages were incapable of running an automated coverage report. Second, 51 repositories (45+6) had `testthat` folders, with a main `testthat.R` file, but did not include any tests.

As a result, 71 R packages were filtered because of low-quality testing by not using automated unit testing frameworks, or having incomplete folders that disabled automated checks. Though the following analysis was completed with the 177 repositories that did have automated unit testing, the reason to filter these 71 packages is indicative of low testing quality (RQ1) and two crucial testing smells (RQ2): lack of unit testing, and manual testing.

A. Testing Quality and Smells (Part I - RQ1, RQ2)

The following subsections discuss all the analysis conducted over the packages, to answer RQ1 and RQ2. The findings and implications relevant to R packages are later summarised on Sections IV-A and IV-B.

1) Testing Coverage: Code coverage analysis reveals the areas not exercised by a set of test cases [15]. There is no proven direct relation that a complete test coverage guarantees a defect-free code [19]. Furthermore, focusing only on test coverage instead of logic being tested is also a source of TTD [5]. Nonetheless, coverage remains a useful tool when analysing testing quality.

This first analysis used the total coverage calculation given by `covr`, and additionally classified repositories by discipline and type. This was done by reading their "readme.md" and description files. The types identify what tools or algorithms are provided by the package.

Figure 2 summarises the averages. The mean total coverage is 48.3%, with the largest distribution of packages having values between 14% and 80%. Overall, the coverage of bioinformatics packages varies dramatically by type, and software-oriented packages (which are just tools or data parsing) are below the mean. This can potentially indicate that few methods are thoroughly tested.

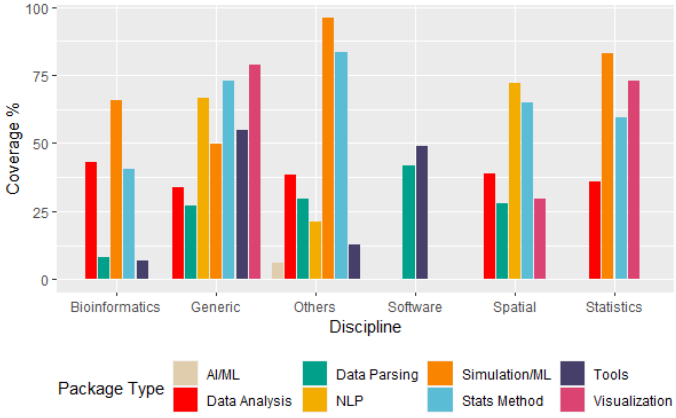

Fig. 2. Average coverage in selected repositories by discipline and type.

2) Relevancy vs Tested Lines: Selected packages included 886738 lines of code (LOC), of which about 40% were relevant (i.e. R code, without comments, empty lines, or similar). `covr` was used to evaluate the proportion of relevant lines that were tested (see Table III). Overall, only 43% of relevant lines are tested.

To explain these results, two in-depth analysis paths were followed. Their goal was to understand a) what is written in the untested relevant lines, and b) what proportion of those lines belongs to non-exported methods; the latter can be thought as the R-equivalent of OO's private methods. This is discussed in the following subsections.

TABLE III
Tested and untested relevant lines in selected repositories.

	Irrelevant Lines	Relevant Lines
Tested Lines		154543
Untested Lines		204652
Total	*527540*	*359198*

3) Untested Lines Analysis: A sub-study was designed to analyse what is written in the untested relevant LOCs, and if those lines belonged to exported or non-exported methods. R's non-exported functions are equivalent to OO's private functions, as they can only be used internally by the package.

Not testing private functions is a conflicting topic discussed among practitioners: it often implies a violation of the single responsibility principle, requiring the use of techniques such as reflection [5]. In this line of thought, private methods are not testing "directly" but "indirectly" through the methods that call them —it is not that 'they are not tested', rather than 'they are tested indirectly'. The analysis of this manuscript adopts this position, recommending an indirect test of private/non-exported methods or functions.

There are two ways for exporting methods in R: i) using

`roxygen2 @export` annotation[5], or ii) labelling as not-exported all functions whose name starts with a given syntax pattern. Though the former is widely accepted and writes a list of exported functions in the package's "namespace" file, the latter is also common. A custom R script mined the "namespace" file of all packages to obtain the names of exported functions, covering option (i). To account for functions exported with (ii), the list was revisited in a second step, automatically adding functions exported by name pattern.

This sub-study required classifying LOCs in categories according to their goal: *returns* (stops, breaks and similar lines that stop a behaviour and go back in the callstack), *alternatives* (optional, alternatives or switch blocks), *logs* (lines that printed information, such as warnings, errors, logs), *loops* (R's one-liner loops such as `lapply`, `do.call` and similar), *wrangling* (lines that allow reorganising data), and *online operations* (such as getting API tokens, and similar).

For the 177 repositories, the dataset of untested relevant lines consisted of 558588 cases. It was not possible to analyse this automatically since there was no pre-classified data, and manually sorting the whole dataset was infeasible due to time constraints. For this reason, a representative sample of untested relevant LOCs (URLOC) was manually classified.

A conservative sample size calculation was performed, by selecting from 5 categories, with a margin of error of 3 percent at 95% confidence using the approach of Thompson [20]. Based on this calculation, a random subset of 1416 URLOCs were sampled from the 558588 and then manually classified by looking at the source code in the specific line. Hence, the results generated from this sub-sampling are considered, at least to some extent, generalisable.

Both datasets (manual classification in types, and list of exported functions by package) were crossed, to obtain the proportion of exported and non-exported URLOCs by type (Figure 3). Overall, 55.3% of URLOCs belong to non-exported functions. The most representative group are *specific alternatives* (32.6%); this is quite negative for TTD, as it highlights a smell since not all paths are appropriately tested. Other relevant groups are *variable preparation* and *wrangling*; these represent an essential operation for data science, as organising and cleaning data is a critical activity [21]. Such a high number of URLOCs in these groups points to another TTD smell, lowering the testing quality.

4) Informative Asserts: Assertions are critical elements for unit testing, as they evaluate whether a predicate is true or false; it compares an expected result to the actual value obtained by the method under testing [15]. An assert "passes" when expected and actual are equal. Though not-passing asserts symbolise bugs, the opposite does not guarantee a defect-free system [22]. *Asserts* are positioned inside of *test methods*.

Furthermore, most unit testing tools offer the ability to print a custom message in assertions, test methods, or both. Confusing, ill-worded messages are a source of TTD and identified as a testing smell [5].

R's most common unit testing package, `testthat` locates messages in the test method signature. An R script pre-classified the signatures, determining that 98% of them had a written text message, while less than 0.5% of test methods stored the messages in a variable. Regarding the former, another script counted the length in words and determined that half of them used between 3 and 7 words per message.

Nonetheless, a second manual sub-study was conducted to analyse the clarity of those messages subjectively. Once again, given the size of the complete dataset, a sub-sample was calculated and extracted. This used the same procedure detailed in Section III-A3, also following Thompson [20]. Due to this, results are also considered somewhat generalisable.

[5] `roxygen2` is a package that provides R with functionalities similar to those of Javadoc, mixed with behaviour from Java annotations

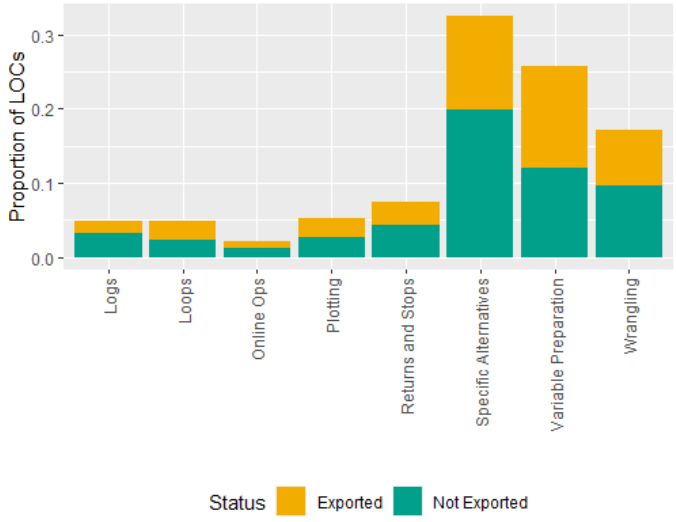

Fig. 3. Tested and untested relevant lines in selected repositories.

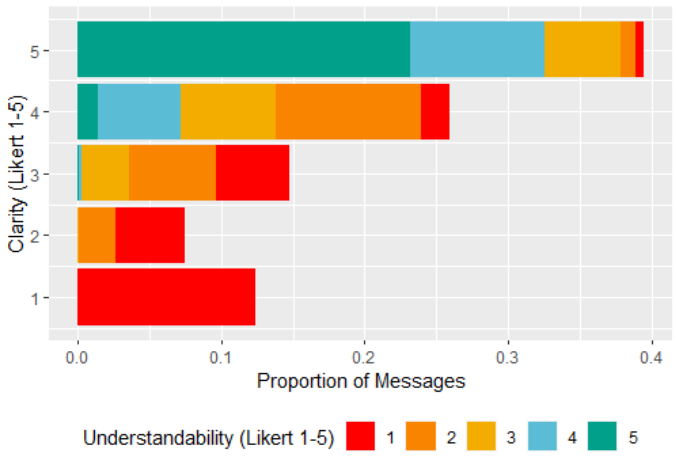

Fig. 4. Clarity and understandability of messages in test methods. Clarity refers to the use of natural language semantics, while understandability concerns grasping what it is being tested

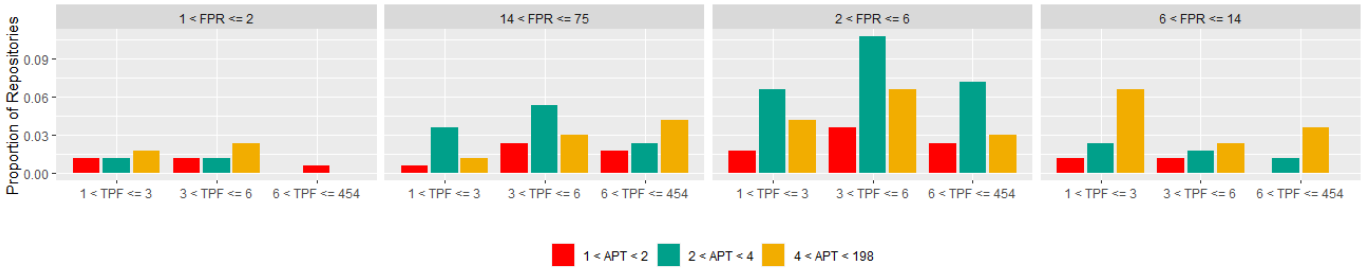

Fig. 5. Proportion of repositories categorised by range of FPR, TPF and APT.

For ten categories, a confidence level of 95% and a margin of error of 3 percent, another sample of 1416 test method messages were randomly extracted from the complete dataset. The ten categories are represented in two Likert scales that indicate *clarity* (in terms of natural language semantics), and *understandability* (recognising what is being tested). The latter requires more technical knowledge of R. Figure 4, compiles results.

Almost 40% messages being labelled *very clear* (Likert 5) and only less than 20% are *very unclear* or *unclear* (Likert 1-2). Nonetheless, about 44.7% are also challenging to understand (Likert 1-2). Furthermore, except for Clarity Likert 5, each category has a larger proportion of mixed-to-lower understandability (Likert 3-1). This is quite negative in terms of TTD, indicating a code smell. Obscure unit tests that are harder to grasp reduce the maintainability of the code.

5) Test Files Organisation : A custom R script was used to crawl the source code of the selected repositories, extracting the signature of test methods, assertions, and LOC position. This was used to analyse the distribution of test methods and assertions per selected repository. Table IV summarises the distributions in quartiles; it is worth noticing that only the first row (FPR) presents data per repository.

TABLE IV
STRUCTURE OF THE SURVEY GENERATED FOR PART II.

Criteria	Code	Q0	Q1	Q2	Q3	Q4
Test files per repository	FPR	1	2	5.5	14	75
Test methods per test file	TPF	1	1	3	6	454
Asserts per test file		1	4	10	21	799
Asserts per test method	APT	1	1	2	4	198

The script identified 9957 test methods in the 177 repositories, with 37166 assertions. This yields a mean of 3.74 assertions per test method, in alignment with the quartiles of APT.

To further this analysis, repositories were classified by range of *file per repository* (FPR), *test methods per test file* (TPF) and *assert per test method* (APT). The ranges were defined using the quartiles identified in Table IV. Therefore, Figure 5 showcases the proportion of repositories for each combination of ranges.

As can be seen, most repositories have between 2 and 5.5 test files, regardless of their size in terms of LOC or R files. In this group, most repositories also include 3-to-6 test methods per test file, with a predominant structure of 2-to-4 assertions in each of them. All groups have a substantial proportion of repositories that have more than one assertion per test method.

Having a single assertion per test method has often been identified as a positive practice; appropriate initialisation methods should be used to avoid repeating preparation process [23]. For example, the JUnit framework for Java handles this with the `@BeforeAll` and `@BeforeEach` annotations. Nonetheless, according to `testthat` 's specification[6], this commonly used framework does not provide means to replicate this initialisation behaviour.

6) Asserts Type and Goals : Asserts are useful to evaluate the internal state of the program. Most unit testing frameworks provide multiple assertion methods to compare different types of actual-vs-expected variables. Assessing which assertions are used the most can help understand what is being tested.

Since all packages analysed used `testthat` as a framework for unit testing, the assertions extracted by crawling the source code of selected repositories were classified as "default" (if provided by `testthat`) or "custom" (if created by the repository's developers). This was checked against `testthat` 's specification and automatically classified through an R script.

Overall, of 37166 assertions detected among all selected repositories, 80.2% were user-defined (custom). There were 102 unique expressions, of which 81 were custom. Furthermore, `testthat` provides 27 default assertions, but only 21 were found in these repositories; those left behind are: `expect_condition`, `expect_vector`, `expect_reference`, `expect_mapequal`, `expect_invisible`, and `expect_visible`.

Figure 6 summarises the popularity of assertions across all of the evaluated repositories, showcasing the total number of uses. Though some custom assertions have multiple uses, this analysis did not validate that implementations across repositories were equivalent.

When designing test cases, good practices indicate that developers should test *common cases* (the "traditional" or

[6]See https://testthat.r-lib.org/reference/index.html

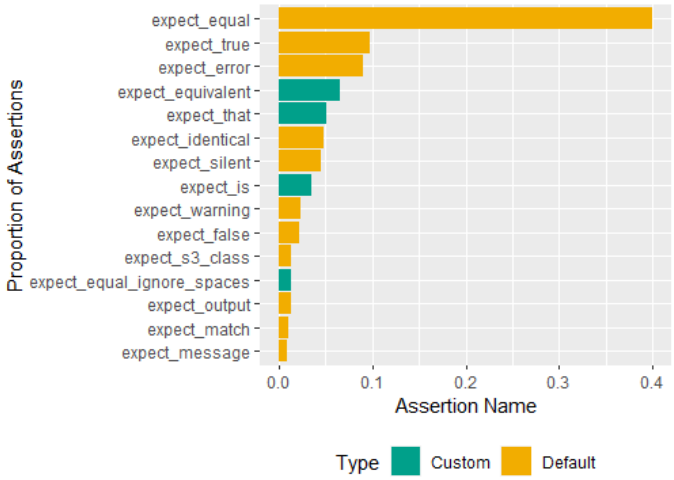

Fig. 6. Top 15 assertions more used in selected repositories, classified as default if offered by `testthat`.

"more used" path of an algorithm or function) as well as *edge cases* (values that require special handling, hence testing boundary conditions of an algorithm or function) [24]. Additionally, *dummy tests* may also appear in the code: these are assertions that "test nothing" (i.e. asserting if a hard-coded true value is true), commonly used to rank up the coverage at the expense of correct testing. Besides those three types, tests can be *automated* (automatically executed and compared using assertions) or *manual* (maybe automatically run, but afterwards it needs a manual check by a person.

Classifying asserts in these groups can highlight quality deficiencies as well as gaps in the unit testing of packages. This was evaluated through a third manual sub-study, that required analysing the code of each assert to classify it in one of the corresponding groups (common, edge or dummy, and automated or manual). Following the same procedure for sub-sampling detailed in Section III-A3, a third sample of 1416 assertions was randomly extracted from the source code of all repositories. As before, results from this analysis are considered to be somewhat generalisable.

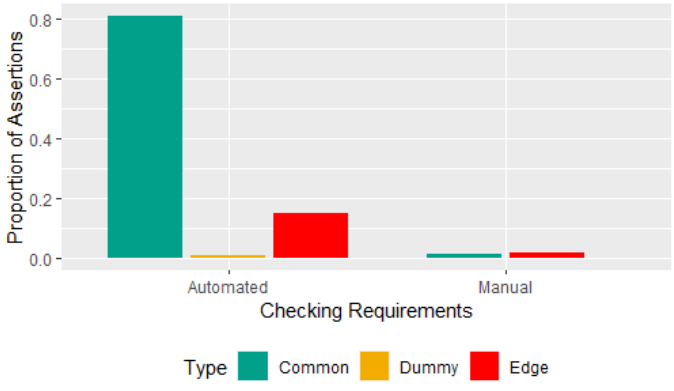

Fig. 7. Manual classification of asserts subsample to determine testing goals.

Figure 7 summarises the results. It is quite negative to see that about 82.5% of the asserts were used to assess *common cases*, disregarding the value of *edge cases*. This indicates the presence of TTD as the latter should also be evaluated since it often generates most bugs; furthermore, it can potentially indicate a lack of exception handling in R code, but more studies are required to assess this further. On a positive note, only 3% of asserts were manual and, upon close inspection, all of them evaluated plots.

B. R Developers Survey (Part II - RQ3)

This section presents the results of the survey conducted on R package developers. Structure, preparation and distribution of the survey were discussed in Section II-C.

Figure 8 summarises the answers to the demographic questions. About half respondents have between 5-10 years of experience as R developers, with almost 27.5% having more than ten years. Furthermore, about 29.7% had between 5-10 R packages published, and 23% had more than ten packages. This indicates that the respondents are mostly very experienced developers, and it is safe to assume they are well-versed in R programming.

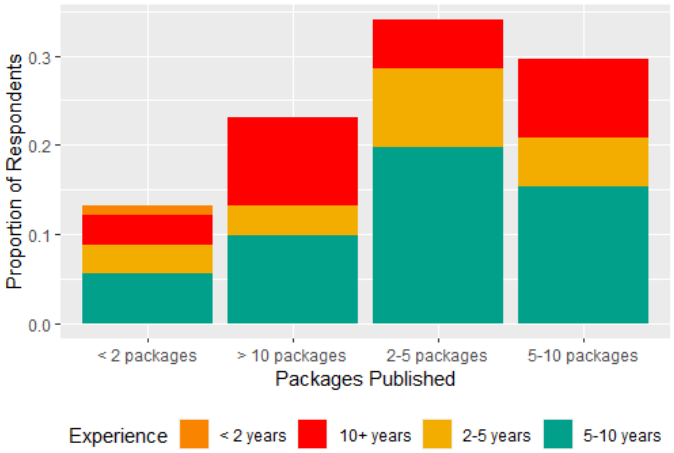

Fig. 8. Self-reported demographics of survey participants.

1) Testing Practices : Regarding testing practices, about 86.8% said they test their code using "unit testing packages", while 12% indicated they only test manually. This is problematic, as manual testing indicates an inappropriate structure, incurs in automation debt and is unable to work as efficiently as unit testing. It could also potentially indicate a lack of functionalities provided by standard R tools. Regarding the latter, 54.9% of respondents declared using `testthat`; other unit testing packages had less than 1% of use.

Table V summarises answers regarding the type of testing performed; it is positive to see that developers are interested in assessing functions individually, but also in testing the package as a whole.

About 58% of respondents replied to the question about the desired improvement to existing tools. The most common

TABLE V
TYPE OF TESTING PRIORITISED BY DEVELOPERS.

Testing Type	Answers %
Unit, evaluating functions individually	50.9%
Integration, studying clusters of functions	28.1%
Systems, using my package externally	14%
Other	1.3%
N/A	5.7%

reply (though worded differently) was "better documentation, tutorials and examples". Due to space limitations, the following is a list of highlighted replies; the full table of replies to this answer is available as supplementary material[7].

- "Honestly anything that would make it faster or automate bits of it. I do a terrible job of testing, and it is largely because it just takes so much time for so little immediate benefit".
- "1- A document that would explain how to create meaningful/efficient unit tests (i.e. beyond just boosting the code coverage). 2- I only use `testthat`, maybe a comparison with the different testing packages would be helpful".
- "Understanding of how to build tests for data science (and data tables generally)".
- "Easier ways to generate test data that's small and easy to include in a package, easy documentation of test result changes to track function changes over time in a documented way."

2) Testing Challenges : Figure 9 summarises the challenges screened in the survey and their reported severity. Similar to other languages, *time constraints* is the most popular limitation, followed by *emphasis on development rather than testing*. On the positive side, few respondents (less than 22%) have trouble discerning the benefits of unit testing tools.

Fig. 9. Severity of challenges, as reported by developers, faced regarding to testing R packages.

However, almost 16% declared *lack of testing experience* as a Serious/Very Serious issue, with similar severity regarding to *poor documentation*. Furthermore, almost 10% still face a *steep learning curve* for unit testing. This is especially concerning given that participants self-reported a high level of expertise in R package development (see Figure 8). If senior developers still face these challenges at this stage of their careers, junior developers may find this even more taxing.

Though overall, 20-40% of respondents do not currently face any of the surveyed challenges, between 22%-49% still deal with them at different degrees of severity.

Furthermore, a typical example of incorrect testing practices is when a unit testing suite passing all test cases, but developers still find bugs in the code when using the package. Close to 58% participants said this happened to them "more than one time", and almost 22% estimated "at least once"; almost 9% "did not remember". Though the option of "this never happened to me" was available, none of the participants chose it.

Though this is quite negative, it aligns with the results obtained in Part I of this study. The extensive focus on testing common cases enables test suites that successfully pass all unit testing, but that do not wholly test the code. This supports the existence of TTD with regards to inadequate unit testing.

3) Coverage Practices: The survey questioned participants about their reliance on coverage tools. About 39.5% said they use them occasionally. About 25% said they "always" use them, and another quarter replied the opposite ("never"). Regarding coverage tools, 24% agreed on using Codecov[8], while 26% used the `covr` R package.

The last question evaluated how developers perceive the coverage of their packages (see Table VI). As can be seen, many participants could be understanding that greater coverage implies better testing (answers like being confident, and bug-free), when this is not always correct. This is demonstrated by a large number of tests suites failing to find bugs and the lack of assertions evaluating edge cases.

TABLE VI
SELF-REPORTED EFFECTS OF COVERAGE VISUALISATIONS.

Coverage Perception	Answers %
It motivates me	30.8%
It makes me more confident in my code	23%
It makes me anxious (there is work left)	5.5%
I trust my code is bug-free	2.2%
Other / None of the above	27.5%
NA	11%

IV. DISCUSSION

This section discusses the answer to the RQs, and results obtained from both parts of the study.

A. RQ1: Quality Testing

This section answers question RQ1, regarding the correctness of the tests found in R packages.

[7]See: https://tinyurl.com/y9ymuv8j

[8]https://codecov.io/

First of all, it is worth noticing that during the repository filtering, 71 packages were separated due to low-quality testing: 20 of them had manual testing only, and the remaining 51 had empty tests. Furthermore, though high coverage values cannot ensure that the code is defect-free [19], there is high variability in the coverage of packages, with some groups having as little as 10%.

Second, only about 43% of relevant lines are tested, but code inspections showcased that alternative paths are the least explored and covered when unit testing R packages. Moreover, the developers' survey confirmed that most of them continue to face test suites with all test passing, yet still find bugs in their code.

Furthermore, R unit testing tools may be incomplete. In particular, though some provided asserts are widely used, others are not, leading many developers to define their custom asserts. Related to this, the most common tool, `testthat` (see Section III-B1) does not provide functions for automated initialisation of test data (see Section III-A5). As a result, this may also hinder the quality of unit testing in R packages.

Overall, it is concerning to see that the unit testing of R packages cannot be considered comprehensive or high-quality. This can be linked to the developers' appraisal of potential improvements for existing tools: better documentation, tutorials and examples, including ways to generate small test data.

B. RQ2: TTD Smells

Smells are "symptoms of poor developing choices", and are related to each type of technical debt [26]. In particular, TTD smells have been previously classified and studied by other authors, providing a well-accepted taxonomy.

As a result, this section answers RQ2 using the information and conclusions extracted from both parts of this study. It organises current testing concerns into the TTD classifications proposed by Samarthyam et al. [5]. Table VII summarises problematic testing types, testing smells, and the results of the study that support it.

Overall, three groups of TTD smells were identified in the mined R packages:

- The most common group are *Unit Testing* smells. These are inadequate cases (testing few paths, with few relevant tests), obscure tests (under the premise of unit testing as live documentation), and improper asserts (non-optimal usage of asserts, focus on coverage rather than common/edge cases, and so on).
- After that, issues in *Exploratory Testing* are also present due to inexperience testers. Results here are related to self-reported challenges regarding unit testing documentation and learning materials (survey results).
- Finally, *Manual Testing* smells are present due to limited test execution (manual testing demands more resources and time, also identified as challenges), and improper test design (requiring manual confirmation and reporting).

C. RQ3: Developer Challenges

The main goal of Part II of this study, the anonymous survey, was to understand developers' subjective perception of testing and the challenges they face. As presented in Section III-B (see Figure 9), the most common concern is regarding *time constraints*, as most developers cannot invest more time in testing, but rather develop the package quickly. Furthermore, different organisations and employers *emphasise development rather than testing*, reasserting this issue.

TABLE VII
TYPES OF TTD, SMELLS, AND RESULTS SHOWCASING WEAK-SPOTS

Type	Smell	Reason
Unit Testing	Inadequate Unit Tests	Elevated number of relevant lines still untested (see Table III and Figure 3).
		Many alternative paths, belonging to exported functions, are not being tested (see Figure 3).
		Elevated variability of coverage between packages of the same discipline. This may indicate incomplete or excess testing (see Figure 2).
		Increased focus on testing common cases, with little focus on assessing edge cases (see Figure 7).
	Obscure Unit Tests	Though many asserts have messages, they are mostly unclear and not understandable (see Figure 4).
		In average, there are too many asserts per test method, lowering the readability of automated testing results (see Table IV and Figure 5).
		Excessive use of custom asserts may hinder testing understandability (see Section III-A6).
	Improper Asserts	Too many common cases are being tested, and few common cases are being evaluated (see Figure 7).
		Excessive use of custom asserts may indicate potential issues with testing frameworks and developers training (see Section III-A6).
		Developers finding bugs regardless of having test suites with all test passing (see Section III-B2).
Exploratory Testing	Inexperienced Testers	Though most survey participants reported a high level of expertise (see Figure 8), their main concern in terms of improvement for existing tools was better documentation, tutorials and examples, as well as guides to create meaningful tests for data science. This is also supported by the indicated severity (medium-to-high) of challenges such as steep learning curve, and poor documentation.
Manual Testing	Limited Test Execution	About 20 papers were filtered as they included only manual testing cases, with no unit testing.
		About 12% of survey participants acknowledged performing only manual testing in their packages (see Section III-B1).
	Improper Test Design	About 3% of asserts were determined to be manual, as they were always testing plots. Though the number is small, there was also a low amount of plotting-related R packages in the selected sample. As plotting and visualisation are vital for data science [25], better testing tools should be developed.

Nonetheless, even though participants are quite experienced developers, the *steep learning curve* and *poor documentation of testing tools* continue to be a challenge. This is coupled to desired improvements in current testing packages, where better documentation and tutorials were the most common request.

It is safe to assume that common challenges can be linked to two potential sources:

- *Lack of training* in developers. Besides self-reported issues on the survey carried out in this study, previous research also demonstrated that most R programmers come from diverse technical backgrounds not focused on programming [10].
- *Incomplete tools* due to the towering number of custom asserts, challenges such as compatibility issues, and desired improvements such as better automation, test data generation, and comparison between testing suits. This is also supported by the lack of methods that could be used to initialise test data.

D. Threats to Validity

External Validity. These relate to the generalisability of the results. The dataset consisted of 177 systematically-selected, open-source R packages mined from GitHub. Still, it is unclear if the findings would generalise to all R packages. Furthermore, the survey respondents sample may not be representative of the entire population of developers, and thus results might not generalise to all of them. Attempts to minimise these biases include surveying a large number of R packages (almost 200), filtered by maintenance conditions, with a large sample of developers. To the best of the author's knowledge, so far, this is the first and largest study regarding unit testing in R packages.

Internal Validity. These concern the conditions under which experiments are performed. Though packages were systematically selected, 31 of them had to be discarded due to being unable to download all required dependencies and run the automated analysis (see `covr`'s limitations, in Table I). Furthermore, since there was no baseline data to use as training sets for automated algorithms, a set of analysis was conducted on randomised sub-samples of data. Though the numbers were statistically defined, they may not be entirely generalisable.

It is worth noticing that the manual classification conducted in Section III-A (i.e. *Analysis of Untested Lines* and *Informative Asserts*) was completed by a single author, which may lead to researcher bias. In each sub-study using this approach, this threat was minimised by completing the classification twice: after the first attempt, the author repeated the process. There was a difference of a week between finishing the first attempt and commencing the second. For the *untested lines*, there were only 17 disagreements, solved by revising the classification; the resolution method used in *Informative Asserts* implied averaging the numbers of both attempts (only for the disagreements).

V. Conclusion

R is a package-based programming ecosystem, mostly targeted to statistics and data science, that provides a simple way to install third-party code and extend the language's functionalities. As a result, defects and bugs present in R packages can transitively affect all packages and scripts that depend on it, effectively becoming a threat to their validity. Testing technical debt (TTD) has been identified as shortcuts (non-optimal decisions) related to testing, that may reduce the quality of the code produced.

This study reports the current state of testing of almost 200 systematically-selected, open-source R packages available in GitHub. The source code of the repositories was analysed to determine the quality of testing and possible smells standard across most R packages. Furthermore, this study also surveyed the developers of these packages to understand the testing culture and determine the challenges they face while testing.

The findings can be summarised as follows:

- R package testing cannot be considered comprehensive or high-quality. Several reasons support this: many alternative paths are not being tested, there is a highly variable coverage, and the occurrence of manual testing.
- Several TTD smells have been identified by comparing the results of the study to existing TTD smells classifications. Common smells are: inadequate and obscure unit tests, improper asserts, inexperienced testers, and improper test design.
- R packages developers face numerous challenges. Participants of the survey self-reported a high level of expertise. However, they agreed on the following challenges: time constraints, emphasis on development rather than testing, poor documentation of tools, steep learning curve, and still finding bugs despise of having test suits with all-passing tests.

This study is the first exploratory step to understand unit testing practices in R package programming. This is considerably novel because R is a multi-paradigm language that combines lazy functional features with minimal object-oriented (OO) features, setting it apart from current OO-focused knowledge of technical debt.

In the future, the authors would like to expand the study by analysing more repositories and surveying more developers. Automated tools can be developed using the manually classified data that was generated in this study, opening the door to large-scale studies. Also, this study focused on R packages but not in R scripts; the latter are a completely different way of programming in R, fundamental to data science based in this language. As a result, an empirical study on common practices and basic technical debt would be required before addressing TTD in this context. Finally, there are also three lines of work related to the results obtained from this study:

1) The automated code analysis identified a large number of irrelevant code lines, with a majority of them potentially linked to comments. Analysing these lines could highlight

the presence of admitted technical debt (either generic or TTD).
2) There is a high variability of coverage in the mined R packages. Though the ideal coverage has been studied in OO-centric programming languages, this number may vary dramatically for R. Further studies could address this area.
3) The study showcased a large amount of non-exported functions being tested. Though this concept is somewhat similar to private methods in OO programming, it is not precisely the same. The impact of testing (or not testing) non-exported functions also requires further analysis.
4) Common unit testing tools used for R packages are incomplete, lacking means to handle a systematic initialisation or cleanup of elements to be tested, thus effectively adding complexity to the process.

Ethical Considerations

The methodology used in this manuscript, and described in the sections below, was approved by RMIT University Human Ethics Research Committee (HREC), with project code 2020-22968-10378.

Acknowledgements

The author thanks Associate Professor John Ormerod, from the School of Mathematics and Statistics at University of Sydney (Australia), for his assistance when calculating the sample sizes for the manual analysis of the data extracted. The author also gratefully acknowledges the anonymous reviewers for their careful reading of the manuscript and their insightful comments and suggestions, as well as the R developers who participated in the survey.

References

[1] R. Muenchen, "The popularity of data analysis software," 2015. [Online]. Available: http://r4stats.com/articles/popularity/

[2] F. Morandat, B. Hill, L. Osvald, and J. Vitek, "Evaluating the design of the r language," in *ECOOP 2012 – Object-Oriented Programming*, J. Noble, Ed. Berlin, Heidelberg: Springer Berlin Heidelberg, 2012, pp. 104–131.

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

[4] W. Tracz, "Refactoring for software design smells: Managing technical debt by girish suryanarayana, ganesh samarthyam, and tushar sharma," *SIGSOFT Softw. Eng. Notes*, vol. 40, no. 6, p. 36, Nov. 2015. [Online]. Available: https://doi.org/10.1145/2830719.2830739

[5] G. Samarthyam, M. Muralidharan, and R. K. Anna, *Understanding Test Debt*. Singapore: Springer Singapore, 2017, pp. 1–17. [Online]. Available: https://doi.org/10.1007/978-981-10-1415-4_1

[6] A. Mueller, "Unit testing in r," 2019. [Online]. Available: https://towardsdatascience.com/unit-testing-in-r-68ab9cc8d211

[7] F. Palomba and A. Zaidman, "Notice of retraction: Does refactoring of test smells induce fixing flaky tests?" in *2017 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, 2017, pp. 1–12.

[8] V. Garousi, B. Kucuk, and M. Felderer, "What we know about smells in software test code," *IEEE Software*, vol. 36, no. 3, pp. 61–73, 2019.

[9] A. Qusef, M. O. Elish, and D. Binkley, "An exploratory study of the relationship between software test smells and fault-proneness," *IEEE Access*, vol. 7, pp. 139 526–139 536, 2019.

[10] D. M. German, B. Adams, and A. E. Hassan, "The Evolution of the R Software Ecosystem," in *2013 17th European Conference on Software Maintenance and Reengineering*, Mar. 2013, pp. 243–252, iSSN: 1534-5351.

[11] A. Decan, T. Mens, M. Claes, and P. Grosjean, "When github meets cran: An analysis of inter-repository package dependency problems," in *2016 IEEE 23rd International Conference on Software Analysis, Evolution, and Reengineering (SANER)*, vol. 1, 2016, pp. 493–504.

[12] C. Ramirez, M. Nagappan, and M. Mirakhorli, "Studying the impact of evolution in r libraries on software engineering research," in *2015 IEEE 1st International Workshop on Software Analytics (SWAN)*, 2015, pp. 29–30.

[13] H. Borges and M. T. Valente, "What's in a github star? understanding repository starring practices in a social coding platform," *Journal of Systems and Software*, vol. 146, pp. 112–129, 2018.

[14] J. Jiang, D. Lo, J. He, X. Xia, P. S. Kochhar, and L. Zhang, "Why and how developers fork what from whom in github," *Empirical Software Engineering*, vol. 22, no. 1, pp. 547–578, Feb 2017. [Online]. Available: https://doi.org/10.1007/s10664-016-9436-6

[15] J. Lawrence, S. Clarke, M. Burnett, and G. Rothermel, "How well do professional developers test with code coverage visualizations? an empirical study," in *2005 IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC'05)*, 2005, pp. 53–60.

[16] J. De Bleser, D. Di Nucci, and C. De Roover, "Assessing diffusion and perception of test smells in scala projects," in *2019 IEEE/ACM 16th International Conference on Mining Software Repositories (MSR)*, 2019, pp. 457–467.

[17] Y. Lu, X. Mao, Z. Li, Y. Zhang, T. Wang, and G. Yin, "Internal quality assurance for external contributions in github: An empirical investigation," *Journal of Software: Evolution and Process*, vol. 30, no. 4, p. e1918, 2018, e1918 smr.1918. [Online]. Available: https://onlinelibrary.wiley.com/doi/abs/10.1002/smr.1918

[18] F. Křikava and J. Vitek, "Tests from traces: Automated unit test extraction for r," in *Proceedings of the 27th ACM SIGSOFT International Symposium on Software Testing and Analysis*, ser. ISSTA 2018. New York, NY, USA: Association for Computing

Machinery, 2018, p. 232–241. [Online]. Available: https://doi.org/10.1145/3213846.3213863

[19] A. Bertolino, B. Miranda, R. Pietrantuono, and S. Russo, "Hybrid is better: Why and how test coverage and software reliability can benefit each other," in *Web Information Systems and Technologies*, M. J. Escalona, F. Domínguez Mayo, T. A. Majchrzak, and V. Monfort, Eds. Cham: Springer International Publishing, 2019, pp. 25–38.

[20] S. K. Thompson, "Sample size for estimating multinomial proportions," *The American Statistician*, vol. 41, no. 1, pp. 42–46, 1987. [Online]. Available: http://www.jstor.org/stable/2684318

[21] F. Endel and H. Piringer, "Data wrangling: Making data useful again," *IFAC-PapersOnLine*, vol. 48, no. 1, pp. 111 – 112, 2015, 8th Vienna International Conferenceon Mathematical Modelling.

[22] R. Khalid, "Towards an automated tool for software testing and analysis," in *2017 14th International Bhurban Conference on Applied Sciences and Technology (IBCAST)*, 2017, pp. 461–465.

[23] G. Meszaros, *xUnit Test Patterns: Refactoring Test Code*, ser. Addison-Wesley Signature Series. Upper Saddle River, NJ: Addison-Wesley, 2007. [Online]. Available: https://www.safaribooksonline.com/library/view/xunit-test-patterns/9780131495050/

[24] E. Daka and G. Fraser, "A survey on unit testing practices and problems," in *2014 IEEE 25th International Symposium on Software Reliability Engineering*, 2014, pp. 201–211.

[25] M. Molina-Solana, D. Birch, and Y. ke Guo, "Improving data exploration in graphs with fuzzy logic and large-scale visualisation," *Applied Soft Computing*, vol. 53, pp. 227 – 235, 2017.

[26] M. Tufano, F. Palomba, G. Bavota, M. Di Penta, R. Oliveto, A. De Lucia, and D. Poshyvanyk, "An empirical investigation into the nature of test smells," in *Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering*, ser. ASE 2016. New York, NY, USA: Association for Computing Machinery, 2016, p. 4–15. [Online]. Available: https://doi.org/10.1145/2970276.2970340