# JTeC: A Large Collection of Java Test Classes for Test Code Analysis and Processing

Federico Corò, Roberto Verdecchia, Emilio Cruciani, Breno Miranda, Antonia Bertolino

# JTeC: A Large Collection of Java Test Classes for Test Code Analysis and Processing

Federico Corò[1,6], Roberto Verdecchia[2,6], Emilio Cruciani[3,6], Breno Miranda[4], Antonia Bertolino[5]

[1]Sapienza University of Rome
[2]Vrije Universiteit Amsterdam
[3]Inria Sophia Antipolis Méditerranée, I3S
[4]Federal University of Pernambuco
[5]Consiglio Nazionale delle Ricerche
[6]Gran Sasso Science Institute

## Abstract

The recent push towards test automation and test-driven development continues to scale up the dimensions of test code that needs to be maintained, analysed, and processed side-by-side with production code. As a consequence, on the one side regression testing techniques, e.g., for test suite prioritization or test case selection, capable to handle such large-scale test suites become indispensable; on the other side, as test code exposes own characteristics, specific techniques for its analysis and refactoring are actively sought. We present JTeC, a large-scale dataset of test cases that researchers can use for benchmarking the above techniques or any other type of tool expressly targeting test code. JTeC collects more than 2.5M test classes belonging to 31K+ GitHub projects and summing up to more than 430 Million SLOCs of ready-to-use real-world test code.

# 1 Introduction

Test automation has been actively pursued since the 90's [17] as a solution to reduce the high costs of software testing and improve product quality [12]. In more recent years, the advent of test driven development [10], and the following broad industrial take-up of continuous integration [5] and DevOps [20] practices, with their promise of faster time to market and improved maintainability, are further pushing companies into test automation. A recent study by Zion Market Research [18] estimates that the software test automation market will grow from the 16 Billion Dollars of 2016 up to 55 Billion Dollars by end of 2022.

However, this insurgence of test automation also comes with challenges and risks, which several researchers have been prompt to identify and face. One of the most evident issues is related to the growing scale of the test code that needs to be maintained, analysed, and processed side-by-side with production code. Indeed, as a result of promoting the practice of testing often and continuously, the sizes of test suites that have to be managed grow to limits that make traditional testing methods and tools not applicable anymore [14]. Therefore, a thread of software testing research is addressing expressly the challenge of finding more efficient and scalable approaches [15, 4, 14]. In [15] we coined the term *big testsets* to denote test suites whose dimensions go beyond the capacity of existing testing tools.

On the other hand, many of the problems that Software Engineering research has been facing for decades in handling large scale software applications re-propose themselves for tackling such big testsets. However, studies have shown that test code exposes different characteristics than production code [19, 6] and hence needs *ad hoc* methods. As a consequence several researchers analyse the quality of test code, propose metrics to assess test code quality, and try to identify useful patterns for good test code.

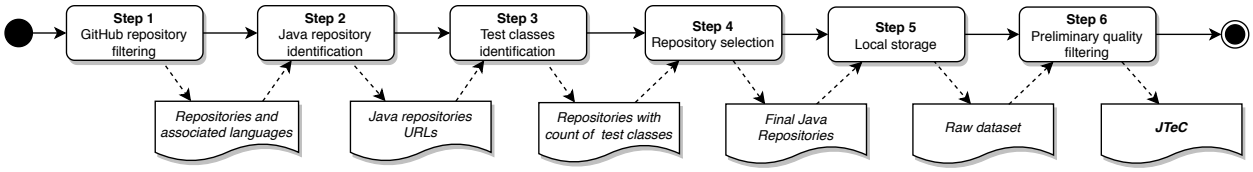


Figure 1: Overview of the JTeC composition steps and intermediate outputs

All the above research efforts demand availability of a large dataset of test code, to which the proposed methods and tools can be applied. However there does not yet exist such a collection for ready usage by the community. Therefore, most studies search in databases of Open Source projects, for example from GitHub, to collect a proper amount of test code: we did this recently in [3] for challenging the efficiency of a scalable test reduction approach, while Gonzales and coauthors [7] made the same for studying testing patterns that can affect maintainability. In the above and many similar studies a consistent amount of effort has to be spent by the authors for setting up the experimental subjects. Moreover, as each team makes its own selection, it is difficult to make a comparison across similar studies.

As we ourselves are going to need such a dataset for validating our scalable approach to test prioritization [15], we would like to make available for the usage of the community the test classes dataset that we collected. At the time of submission, the dataset, called JTeC (Java Test Classes), provides 2.5M+ test classes collected from a set of 31K+ projects in Github, which from now on the community working on test code analysis and processing can reuse without spending further effort. The JTeC dataset, and the quality filtering script which is provided as complement to the dataset, are available online in the **JTeC bundle**.[1]

In the remainder we describe the methodology followed to collect the dataset (Section 2), its structure (Section 3), and how it can be used for research (Section 4). We conclude the paper hinting at envisaged future JTeC developments.

[1] `https://doi.org/10.5281/zenodo.3711509`

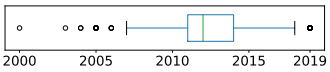


Figure 2: Distribution of dates of considered commits

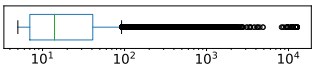


Figure 3: Distribution of test classes per repository

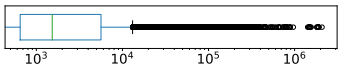


Figure 4: Distribution of SLOC per repository

## 2 Methodology

In this section we report the methodology we use to gather the data of our dataset. The process, illustrated in Figure 1, consists of six main steps: (i) GitHub repository filtering, (ii) Java repository identification, (iii) test classes identification, (iv) repository selection, (v) local storage of test classes, and (vi) preliminary quality filtering. In the reminder of this section we document the specifics of each step. For lower-level implementation details, we refer the reader to our replication package, containing the entirety of the scripts utilized to generate the dataset[2] for the sake of the verification of the dataset, its replication and extension.

**Step 1: GitHub repository filtering.** The first step of our process consists of indexing the public GitHub repositories, and is carried out in order to execute efficiently the subsequent phases of our process. In this step we first retrieve the name of the public repositories and the username of their creators. This is achieved via a query to the GitHub API[3] specifying the unique identifiers (ID) of the repositories in an incremental fashion (starting from the repository with ID=1). As stopping criterion for the repository indexing process we adopt the number of repositories required to collect 2.5M test classes. This number is approximated via a preliminary exploratory analysis, and amounts to approximately 31K repositories. We chose to target 2.5M test classes as we deem such number adequate in order to carry out studies requiring as input a large scale amount of heterogeneous test source code.

Subsequently, once we obtain the list of repository names mapped to the usernames of their creators, we can launch a second query to the GitHub API in order to retrieve the programming languages associated with each repository. The final output of this step consists of a local .csv file containing for each public repository indexed the following fields: *repository ID*, *username of repository creator*, *name of the repository*, and *programming languages associated to the repository*.

**Step 2: Java repository identification.** Once obtained a local copy of the indexed repositories mapped to their programming languages, we can effortlessly retrieve the URLs of the repositories developed in a specific language. In our case, we parse the .csv file created in the previous step to isolate the repositories developed exclusively in Java by inspecting the language tag of the indexed repositories, and subsequently combine their name with the username of their creators in order to identify the unique URL of the repositories. By parameterizing in our approach the programming language to be considered, we make the process effortlessly modifiable and extensible to take into account other languages, which are not utilized for the creation of the current version of the dataset. The final output of this step is the list of URLs corresponding to all Java repositories out of the ones indexed in Step 1.

**Step 3: Test classes identification.** Once we isolated the repositories developed in Java, we can identify the test classes of the repositories to store them locally. This is achieved by leveraging the standard naming conventions for Java test classes in Junit, the most popular Java unit testing framework [13]. Specifically, we do this by selecting the source code files of the selected repositories whose name ends with "Test.java" and "Tests.java".

More accurate methods could be used to select test cases, e.g., via static source code analysis [7]. Nevertheless, we opt for this solution as, given the large scale nature of our dataset, more complex solutions would potentially entail slower running times, hindering our ability to build the dataset in a reasonable amount of time. Moreover, by adopting a standard naming convention to select test classes, we aim not to introduce any significant bias.

[2] `https://github.com/JTeCDataset/JTeC`
[3] `https://developer.github.com/v3`

More in detail, this third step entails the recursive retrieval of the file listing of the repositories[4] by executing a query via the GitHub API. Subsequently, the list of files per repository is parsed in order to identify the files matching our specified naming convention. The naming convention adopted is designed as a parameter in our process, and can hence be adapted in order to consider a different naming convention w.r.t. the one adopted to create the current version of the dataset. The final output of this step consists of a list of Java repositories mapped to the number of test cases they contain, and commit hash considered (for replication purposes).

**Step 4: Repository selection.** Once we gather the list of Java repositories, we select all the original repositories present in the list for their subsequent inclusion in the dataset. In order to enhance the dataset and enable a wider range of potential research studies, we select among the forks of each original repository the one which contains the highest number of test cases. We opt to include exclusively the fork containing the highest number of test cases as this option maximizes the efficiency and effectiveness of the crawling process. Additionally, such a heuristic provides us a fitting criterion to pick, among various forks of a repository, the one which is best fitted to be included in our large scale collection of testing artifacts. As further documented in Section 3, forked repositories can effortlessly be trimmed out the JTeC dataset through a post-hoc filtering process (see Step 6).

**Step 5: Local storage of test classes.** Once we identify the Java repositories to be included in the dataset, we proceed to store locally the source code of the identified test classes. This is achieved by taking advantage of the file listing of the repositories retrieved in Step 3 to identify the test classes. For traceability purposes, the filesystem hierarchy of the repositories is preserved while storing locally the test source code. The output of this process is the entirety of the source code of the identified Java test classes and a .csv file specifying the repositories included, their considered commit hash, and their number of test cases.

**Step 6: Preliminary quality filtering.** To ensure the quality of the data contained in JTeC, we carry out a preliminary quality filtering process on the collected raw dataset. This process is carried out by excluding potential "toy" test suites, and entails identifying test suites comprising less than 5 test cases, and subsequently removing such instances from the dataset. The final output of this step is the final JTeC dataset, which is available online as part of the JTeC bundle [2]. In order to make the dataset further tunable according to specific quality requirements which JTeC users may have, we provide the quality filtering script as complement of the JTeC dataset. Such script, taking as input several variables according to the dataset schema, enables the effortless execution of ad-hoc trimming operations on JTeC, as further documented in Section 3.1.

# 3 Description

The dataset is provided in compressed format and contains the entirety of the collected test source code (`JTeC-Bundle.tar.gz`). The total data storage space of the dataset amounts to 15 GB, and comprises 2.5M+ Java test classes distributed over 31K repositories, summing up to 431M source lines of code (SLOC). As the hierarchical tree structure of the GitHub repositories is preserved, the first two folder levels of JTeC consist of the repository creator username followed by the repository name. Further nested into the filesystem tree reside the test classes collected, according to the original ordering of the test files as reported in the GitHub repositories.

In addition to the complete collection of gathered source code, we provide an overview of the JTeC metadata in the file `JTeC.csv`. Such a file contains the most relevant information for each of the included repositories, in order to provide an eagle-eye overview of the data constituting the JTeC dataset. Specifically, each row of `JTeC.csv` corresponds to one of the repositories contained in JTeC, while the columns represent the following attributes:

- `user`: repository owner,
- `repository`: repository name,

[4]We purposely design this step to consider the default branch and the most recent commit of the repositories, as this will potentially lead to the identification of a higher number of test cases.

- `id`: repository unique incremental identifier provided by GitHub API,
- `fork_id`: unique incremental identifier of the original repository provided by GitHub API (value not empty only for repository forks),
- `hash`: repository commit hash of the version in JTeC,
- `date`: date of the commit considered in JTeC,
- `n_tests`: number of identified test classes,
- `SLOC`: total SLOC of the test classes,
- `size`: total data storage space (Bytes) of the test classes.

An overview of the distribution of the `date` attribute of the repositories stored in JTeC is depicted in Figure 2. From the distribution we can observe that the median commit is dated to late-2012. This trend has to be attributed to the adoption of the repository ID in Step 1 to index repositories. We expect the distribution to be more skewed towards more recent dates if the dataset is expanded in the future, as more recent and active repositories would potentially be included. By inspecting more in depth the data, we can conclude that most of the outliers on the left hand side of the diagram correspond to commit dates modified manually *a posteriori* by the authors, as they are prior to both the creation git and GitHub. We opted not to remove such occurrences, as such process falls outside the scope of our dataset.

The distribution of the `n_test` attribute of the repositories stored in JTeC is depicted in Figure 3. From the figure we can observe that the number of test classes varies highly across repositories, ranging from 5 test cases to more than 12 thousands, with a median of 14 test classes per repository and a mean of 85. Given the heterogeneous nature of this attribute, JTeC constitutes an encompassing dataset of testing artifacts, which enables via the filtering script to effortlessly retrieve source code according to specific ranges of test suite sizes, from small suites to big ones containing thousands of test classes.

In Figure 4, the distribution of SLOC per test suite is reported. As we can observe, the SLOC vary from a few lines per test suite to test suites containing more than 2M test SLOC (including test suites belonging to some projects of popular open source foundations, e.g. Apache and Eclipse). As for the number of test classes collected in JTeC, the variability of the `SLOC` attribute points to the heterogeneity of the source code collected, which encompasses small test suites as well as the ones of prominent open source projects.

## 3.1 Quality filter

As integrating part of the JTeC dataset, we make available in the JTeC bundle [2] the filtering script adopted to carry out the preliminary quality filtering (see Section 2 – Step 6). Through such script we provide JTeC users with the capability to effortlessly explore the dataset and trim it according to their needs by applying further quality filters. The quality filter script, in fact, allows the user to generate a refined version of the dataset that only contains projects and test classes that adhere to user-defined constraints. The user, through a configuration file (`config.json`) provided in the JTeC bundle, can define a range of values for the following criteria (prefix `MIN_`/`MAX_`):

- `TS_Size`: test suite size (measured as number of test classes),
- `TS_Bytes`: total number of Bytes of test suite,
- `TS_SLOCs`: total number of SLOCs of test suite,
- `TS_Year`: test suite year (based on the last commit date).

Additionally, the user has the ability to (prefix `BOOL_`):

- `TS_Clone`: generate a cleaned copy of the dataset in folder `JTeC-Clean/`, with the same structure as that of `JTeC/`,

- `TS_Index`: create `JTeC-Clean.csv`, an index of the cleaned dataset (in the same fashion of `JTeC.csv`),
- `TS_Original`: include original projects in the cleaned dataset,
- `TS_Fork`: include forks with the highest number of test classes (at the moment of download) in the cleaned dataset.

# 4 Purpose and applications

## 4.1 Our motivation

In our previous work we have been investigating scalable approaches for test case prioritization [15] and test suite reduction [3]. In [15], for answering our research question related to scalability, we looked for datasets of big test suites (our aim was to have at least one million test cases) but because such a dataset was not available, we had to rely on the generation of synthetic test cases for carrying out our studies. Similarly, we also evaluated the scalability of the approach proposed in [3]. This time, however, for minimizing possible threats to the validity associated with the use of synthetic tests, we downloaded 500K real test cases from multiple GitHub projects (without following any systematic approach, though). These experiences led us to realize it was time for us to create a curated dataset of test cases that could be used in our own research and shared with the community.

## 4.2 Potential applications

Beyond our own motivation, we consider JTeC could be used to the benefits of researchers in several areas, including:

- **Static analysis**: static analysis of test code has been applied for many purposes, including, for example, the identification of test smells [16], the study of test patterns [7], and recommendations for test code refactoring [19, 9]. JTeC can support such analyses by providing a ready-for-use large dataset of test classes.
- **Regression testing**: JTeC can be used as a benchmark to assess efficiency and scalability of regression techniques that use only test code as input, e.g., black-box similarity based techniques of our own recent work [15, 3].
- **Test case generation**: within the ElasTest project [1], a recommender engine developed by the IBM partner uses machine learning techniques to generate new test cases based on the knowledge acquired by analyzing the existing ones [11]. Supported by JTeC, similar studies can be applied from a per-project perspective, where the idea is to analyze the existing test cases to generate new ones for the same project, as well as for inter-project learning, where the idea would be to learn from multiple sources to generate new test cases to a target project.

# 5 Conclusions and future work

We have presented the JTeC dataset that makes ready available to the community of software testing researchers a large collection of Java test classes useful for several potential purposes related to test code analysis and processing. Our aim goes beyond the current version of the dataset: we strive towards the establishment of a continuously updated dataset, collecting together in a single source the test code belonging to the vast majority of test cases publicly available.

In the future the dataset can be expanded in several directions: as a first step, we intend to augment the test collection by including test code written in other programming languages. Such variation, which is relatively easy to implement (for example by using GHTorrent [8]) will allow the community to answer

interesting research questions not applicable to a purely Java repository. For example, *what are the characteristics of test cases written for projects with different size, complexity, development history, programming languages, etc? What are the common practices of developers/testers when writing test cases in a particular programming language?*

Another important improvement we envisage is to compute and provide test related measurements with the test classes, for example code coverage information. Given the large scale, obtaining such information is clearly highly effort-intensive, and we consider that coarse criteria such as function coverage would be enough. Having this type of information could support benchmarking other regression testing techniques beyond black-box similarity-based ones.

Additionally, to broaden the capabilities of JTeC, we envision to include in future versions also fault information of the identified test classes. Providing this latter type of information, would enable researchers to have at disposal a large-scale real-world dataset containing the most important information related to test suites, enabling them to conduct a vast range of studies related to the research field of software testing.

As a final word, we of course welcome the maintenance support and more ideas for improvement coming from the community over which JTeC is now handed.

# References


[1] ElasTest Project, 2017.

[2] Federico Corò, Roberto Verdecchia, Emilio Cruciani, Breno Miranda, and Antonia Bertolino. JTeC: A Large Collection of Java Test Classes forTest Code Analysis and Processing, May 2019. Companion page for the JTeC dataset at https://github.com/JTeCDataset/JTeC.

[3] Emilio Cruciani, Breno Miranda, Roberto Verdecchia, and Antonia Bertolino. Scalable approaches for test suite reduction. In *Proceedings of the 41st International Conference on Software Engineering*, ICSE '19, 2019.

[4] Sebastian Elbaum, Gregg Rothermel, and John Penix. Techniques for improving regression testing in continuous integration development environments. In *Proceedings of the 22Nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*, FSE 2014, pages 235–245, New York, NY, USA, 2014. ACM.

[5] Martin Fowler and Matthew Foemmel. Continuous integration. *Thought-Works*, 122:14, 2006. Accessed: 2019-01-22.

[6] Vahid Garousi and Michael Felderer. Developing, verifying, and maintaining high-quality automated test scripts. *IEEE Software*, (3):68–75, 2016.

[7] Danielle Gonzalez, Joanna Santos, Andrew Popovich, Mehdi Mirakhorli, and Mei Nagappan. A large-scale study on the usage of testing patterns that address maintainability attributes: patterns for ease of modification, diagnoses, and comprehension. In *Proceedings of the 14th International Conference on Mining Software Repositories*, pages 391–401. IEEE Press, 2017.

[8] Georgios Gousios. The GHTorent dataset and tool suite. In *Proceedings of the 10th working conference on mining software repositories*, pages 233–236. IEEE Press, 2013.

[9] Michaela Greiler, Arie Van Deursen, and Margaret-Anne Storey. Automated detection of test fixture strategies and smells. In *Software Testing, Verification and Validation (ICST), 2013 IEEE Sixth International Conference on*, pages 322–331. IEEE, 2013.

[10] David Janzen and Hossein Saiedian. Test-driven development concepts, taxonomy, and future direction. *Computer*, 38(9):43–50, 2005.

[11] Magda Kacmajor. Say it in plain english, 2018. Accessed: 2019-02-05.

[12] Jussi Kasurinen, Ossi Taipale, and Kari Smolander. Software test automation in practice: empirical observations. *Advances in Software Engineering*, 2010, 2010.

[13] M. Linares-Vásquez, C. Bernal-Cardenas, K. Moran, and D. Poshyvanyk. How do developers test android applications? In *2017 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, pages 613–622, Sep. 2017.

[14] Atif Memon, Zebao Gao, Bao Nguyen, Sanjeev Dhanda, Eric Nickell, Rob Siemborski, and John Micco. Taming google-scale continuous testing. In *Proceedings of the 39th International Conference on Software Engineering: Software Engineering in Practice Track*, SEIP'17, pages 233–242, Piscataway, NJ, USA, 2017. IEEE Press.

[15] Breno Miranda, Emilio Cruciani, Roberto Verdecchia, and Antonia Bertolino. FAST approaches to scalable similarity-based test case prioritization. In *Proceedings of the 40th International Conference on Software Engineering*, ICSE '18, pages 222–232, New York, NY, USA, 2018. ACM.

[16] Fabio Palomba, Andy Zaidman, and Andrea De Lucia. Automatic test smell detection using information retrieval techniques. In *2018 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, pages 311–322. IEEE, 2018.

[17] Bret Pettichord. Success with test automation. In *9th International Quality Week Conference*, 1996. Accessed: 2019-01-22.

[18] Zion Market Research. Test automation market by test type: Global industry perspective, comprehensive analysis, and forecast, 2016 - 2022, 2017. Accessed: 2019-02-04.

[19] Arie Van Deursen, Leon Moonen, Alex Van Den Bergh, and Gerard Kok. Refactoring test code. In *Proceedings of the 2nd international conference on extreme programming and flexible processes in software engineering (XP2001)*, pages 92–95, 2001.

[20] Liming Zhu, Len Bass, and George Champlin-Scharff. Devops and its practices. *IEEE Software*, 33(3):32–34, 2016.