Software Engineering Collaboratories (SEClabs) and Collaboratories as a Service (CaaS)

Elena Sherman
Boise State University
Boise, Idaho, USA
elenasherman@boisestate.edu

Robert Dyer
Bowling Green State University
Bowling Green, Ohio, USA
rdyer@bgsu.edu

ABSTRACT

Novel research ideas require strong evaluations. Modern software engineering research evaluation typically requires a set of benchmark programs. Open source software repositories have provided a great opportunity for researchers to find such programs for use in their evaluations. Many tools/techniques have been developed to help automate the curation of open source software. There has also been encouragement for researchers to provide their research artifacts so that other researchers can easily reproduce the results. We argue that these two trends (i.e., curating open source software for research evaluation and the providing of research artifacts) drive the need for Software Engineer Collaboratories (SEClabs). We envision research communities coming together to create SEClab instances, where research artifacts can be made publicly available to other researchers. The community can then vet such artifacts and make them available as a service, thus turning the collaboratory into a Collaboratory as a Service (CaaS). If our vision is realized, the speed and transparency of research will drastically increase.

CCS CONCEPTS

• **Software and its engineering → Collaboration in software development**;

KEYWORDS

collaboratory, software as a service, research as a service

ACM Reference Format:
Elena Sherman and Robert Dyer. 2018. Software Engineering Collaboratories (SEClabs) and Collaboratories as a Service (CaaS). In *Proceedings of the 26th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE '18), November 4–9, 2018, Lake Buena Vista, FL, USA.* ACM, New York, NY, USA, 5 pages. https://doi.org/10.1145/3236024.3264839

1 INTRODUCTION

One important criterion for evaluating a novel research idea are the evidence supporting the idea's claims. In software engineering, rigorous empirical evaluations commonly serve as such evidence. Moreover, empirical evaluations are almost a de facto standard for scientific contributions in software engineering – it is rare to encounter a software engineering publication with supporting evidence other than empirical experiments. This strong emphasis on evidence-based software engineering research originates from the initiative taken by researchers in the early 2000's to address the concerns of poor reproducibility of empirical experiments, and the weak representativeness of the experimental results due to the deficiency of realistic, standardized benchmark programs [2, 3].

With the emergence of software repositories that offer free hosting services to open-source projects and the increase in the availability of open-source programs, by the mid 2000's researchers established various real program artifact collections to use in empirical evaluations. Examples include SIR [3], which contains program versions, test cases, and execution scripts to support controlled experiments in program testing, DaCapo [1] a benchmark suite that contains realistic object behavior and demanding memory usage, and Qualitas Corpus [18], which aggregates open-source projects from different sources to empirically study the code structure.

While still widely used in empirical evaluations, the aforementioned "static" repositories are becoming exhausted of their relevance since they contain mainly older programs with outdated programming practices and the types of problems they solve. Most importantly, the number of programs in those repositories remains small. For example, SIR, DaCapo, and Qualitas Corpus contain between 14 and 112 programs and were last revised over 5 years ago. In order to keep these repositories current, their programs must be constantly updated and the number of those programs must be increased by at least an order of magnitude. Unfortunately, because of their design the upkeep of static repositories demands an enormous manual effort and dedication of the hosting research groups. Obviously, the fulfillment of such requirements is intractable in the long term and automating the maintenance tasks such as *obtaining* and *curating* the benchmark programs is the only feasible solution.

Automated retrieval of programs from open-source software repositories (GitHub, SourceForge, Bitbucket) has been gaining in popularity and considered the standard in the mining software repository (MSR) community, where researchers analyze program structures and metadata directly from the source code locations. With the support of software mining tools such as Boa [5], MSR researchers use tens of thousands [6, 16, 19] and even hundreds of thousands [13] of programs in their empirical studies. Moreover, researchers in the program analysis community also investigate an automated program retrieval approach [4], which filters programs from GitHub based on project tags like web applications, then downloads the filtered projects and attempts to compile them.

While the process of obtaining programs from open-source repositories has been automated, the most challenging part of automating

ESEC/FSE '18, November 4–9, 2018, Lake Buena Vista, FL, USA

ACM ISBN 978-1-4503-5573-5/18/11…$15.00
https://doi.org/10.1145/3236024.3264839

the curation of those programs remains mainly an open problem. Curation ensures the quality of the obtained programs as well as providing necessary support for the compilation and execution of those programs. The curation efforts depend on the type of software engineering research the benchmark repository supports. For example, curation might involve removing duplicate or similar programs, ensuring that programs can compile, providing a set of test cases for some adequacy criterion, or creating scripts for running the experiments. In the literature there are attempts to apply some level of automated curation: establishing metrics for open-source program repositories [11], identifying similarities between features of obtained programs [14], and obtaining missing dependencies and generating build scripts as in the 50K-C [10] Java project.

After obtaining adequate programs, researchers can perform their empirical evaluations. However, running experiments with those programs might require of researchers a substantial effort to set them up in their experimental environments and to provide sufficient computational power to conduct the experiments. To ensure the reproducibility of the experiments, researchers should detail all the parts of the experiment: the benchmark programs, the experimental set up and the execution environment. With limited space available, researchers might have to sacrifice proper explanation of a novel research approach in favor of detailing experimental set up descriptions. Thus, the current process that researchers use to set up, conduct, and then describe the experiments requires additional effort that is not directly related to advancing software engineering research. Providing means for researchers to directly execute their experiments in a pre-setup, standardized, scalable infrastructure would eliminate this unnecessary burden on the researchers.

In the next section we present our overall vision for a software engineering collaboratory, *SEClab*, where we address the deficiencies of the current empirical evaluation approach identified above. Then, we describe in detail an instance of *SEClab*, designed for a specific software engineering task: static program analysis, which we call *SAClab*. Finally we describe our vision of how different SEClab instances could work together to enable new research directions.

2 VISION OF A SOFTWARE ENGINEERING COLLABORATORY

Unlike a scientific collaboratory that mainly provides instruments to process and share experimental data, a software engineering collaboratory should provide data to evaluate a given instrument. Being conceptually different from existing scientific collaboratories, a different approach is required in its design. We envision that every software engineering task, T, will have a specialized collaboratory, SEClab[T], that enables researchers working on improving T to seamlessly conduct an empirical evaluation of T's advances, e.g., advances in program maintenance, testing, analysis and so on. While each T might require specific instantiations of SEClab[T] components, we present here an overall vision of SEClabs.

Figure 1 depicts the main components of SEClab and its interactions with different stakeholders and resources. The numerical labels on the arrows indicate the order of SEClab operational flow, which the researcher initiates. First, SEClab receives from the researcher the task of type T to be evaluated and additional evaluation requirements (1). Then, using program specifications, SEClab

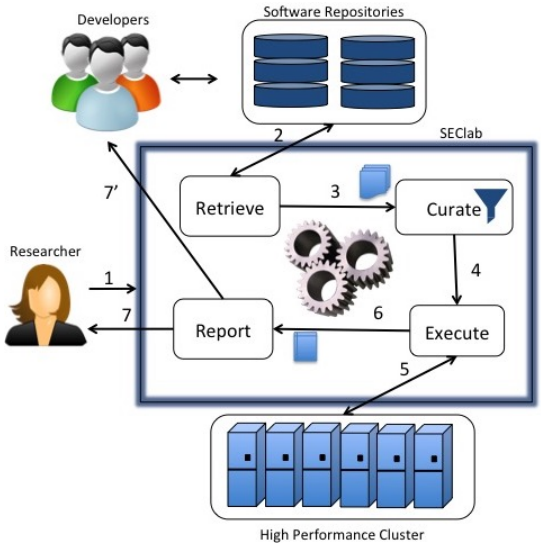

Figure 1: Vision of SEClab

queries open-source repositories and retrieves potentially relevant software projects (2). Upon receiving those projects (3), SEClab uses its curation component to further filter appropriate programs and prepare them for evaluation (4). Given the provided task and the prepared programs, SEClab then performs an empirical evaluation using a computer cluster (5). SEClab then collects and processes the results of the experiments (6) and presents them to the researcher (7) and software repository developers (7'). Below we describe each of SEClab[T]'s main functions in detail.

Request. The researcher initiates the process by supplying SEClab with the task that they want to evaluate. Also, they might provide additional descriptions for setting up the experiments. For example, they might define the type of programs the task should be evaluated on, the Java version, etc. These instructions are expressed in a domain-specific language describing behavioral, structural, and other properties of a program. These requirements are later used for the retrieval and curation of the programs.

Retrieve. The retrieval process uses some experiment set up descriptions to identify potentially suitable projects from software repositories. Preliminary filtering could be done through repository tags [4], or through some previously community-accepted metrics computed from software project metadata [11]. Upon downloading the potential programs, SEClab also creates unique identifiers that track the programs to their locations in repositories.

Curate. The implementation of the curation component is specific to T. Some tasks require no further pre-processing of the retrieved programs or generation of additional supporting artifacts. For example, software engineering tasks used in MSR research require little or no curation, i.e., every potential program retrieved from repositories is adequate. This low level of curation allows MSR researchers to evaluate their hypotheses on hundreds of thousands of programs. Other software engineering tasks demand extensive curation of programs. Compiler optimization tasks require suitable programs to compile. If a potential program cannot be compiled, then the curation process removes such program from further consideration. Besides being compilable, some testing tasks require

programs with test suites. Other examples of curation efforts for different tasks are: program transformation, generating mutants for programs, and producing programs with a certain type of defects. Curation is the most challenging component of SEClab and it is also task dependent, so describing its vision in general terms might be insufficient. Thus, in Subsection 2.1 we describe in detail the required curation effort for static program analysis.

Execute. SEClab performs empirical evaluations according to the researcher's requirements and the practices established by other researchers working on the same software engineering task. For example, the responsibilities of this component might include correctly setting environment, enabling efficient memory profiling, repeating experiments appropriate to ensure a desired confidence level, and resolving abnormal terminations. In order to conduct large scale evaluations, it is imperative that SEClab has access to a high performance cluster.

Report. Reporting completes the researcher's request by providing them with results of the empirical evaluations and also with the description of the evaluations themselves. Later during the results' dissemination, instead of spending time describing the evaluation set up, the researcher can provide the summary and a link to the experiment's detailed description.

SEClab provides useful feedback to the developers whose programs are used in the experiments. For example, if the software engineering task found a program defect, SEClab can report it back to the developers. We hope that by allowing such feedback some developers would allow SEClab to access to their private projects.

2.1 Software Analysis Collaboratory

In this section we describe an instance of SEClab for a Static Analysis task, or SEClab[SA] (aka, SAClab). Research on static program analysis encompasses several types of static analysis. In this presentation we focus on heavy-weight analyses that interpret a program's semantics at the highest level. Symbolic Execution, Predicate Abstraction, and Abstract Interpretation are primary examples of such analyses since they reason about possible values of program variables. These static analysis techniques play an important role in providing the highest software quality assurance and are commonly used to find defects in safety-critical applications. Moreover, researchers from software engineering and programming languages make use of heavy-weight static analysis to advance research in their domains. For example, program optimization researchers use heavy-weight static analysis techniques to improve precision of their medium-weight static analysis [17]. Software testing researchers use symbolic execution to generate an optimal test suite [15] or repair program defects [8, 9].

The challenge with heavy-weight static analysis tools is that they pose additional requirements on a program's structure, such as focusing only on a particular data type, or allowing only intra-procedural analysis. Thus, a researcher might request SAClab to perform evaluations on specific programs. Finding a large quantity of real-world programs that completely match the evaluation requirements might be challenging. The curation component should transform promising potential programs to suitable ones.

In this vision of SAClab we illustrate how program analysis researchers in the programming language and software engineering communities would use the automated services SAClab will provide, and a description of the instances of the components corresponding to the ones in Figure 1. Due to space limitations we focus on thee key functionalities of SAClab: a project discovery subcomponent together with a repository acquisition subcomponent (Retrieve), a program transformation component (Curate), and an analysis execution component (Execute). Each component provides an independent service that researchers can choose to utilize together or use separately. Below, we describe the steps of a typical SAClab session and detail the services each subcomponent provides.

(1) *Researcher provides requirement specifications.* Researchers use a web-based interface to access the infrastructure and specify selection criteria for benchmark programs they seek, along with a description of all required program transformations. Each researcher also has the option to upload the analysis tool under development as a Docker image [7], a framework for light-weight virtualization.

(2) *System searches and locates program candidates.* Project Discovery: using provided program specifications as input, SAClab connects to remote software repositories, such as GitHub, GHTorrent, 50K-C or services such as Boa or RepoReaper, to locate potential candidate projects. SAClab provides predefined selection criteria or allows the researcher to express it through a domain specific language. SAClab can utilize the search options that source locations provide to further narrow the selection process. For example, GitHub allows a user to search by project category, programming language, etc. The output of this step is a list of URLs of candidate projects containing potential benchmark programs.

(3) *System clones and retrieves program candidates.* Repository Acquisition: each time the infrastructure locates potential benchmark program candidates, it connects to the remote repositories to clone the candidate projects, maintaining a local cache for efficiency.

The acquisition subcomponent processes the URL list, which could also be supplied by researchers directly. For each URL the subcomponent clones and arranges the source code of the referenced project. After processing the list it produces a directory structure of cloned code and a log file of successfully and unsuccessfully cloned programs. The infrastructure will allow the user to browse through the obtained code and the log file. Optionally, researchers can also upload their own programs as archives, which the component will copy to SAClab's storage space.

(4) *System modifies programs and documents updates to meet requirements.* Program Transformation: should candidate programs require changes, the system will apply the program transformations that the researcher uploaded, resulting in a set of program variants tailored to meet the benchmarking needs of the researcher's particular analysis tool. To make it easier to reproduce results in the future, the system also maintains a transformation log.

The program transformation component takes as input the path to the candidate benchmark programs directory and transformation specifications. It will serve as a clear, consistent and traceable program transformation instrument. As with program specifications, initially SAClab can allow a researcher to select from a list of predefined transformation specifications or provide additional flexibility in transformation specifications through a domain specific language. The output of the component is a directory with program variants and a log file describing the source of the original program, the transformation applied and the location of the variant.

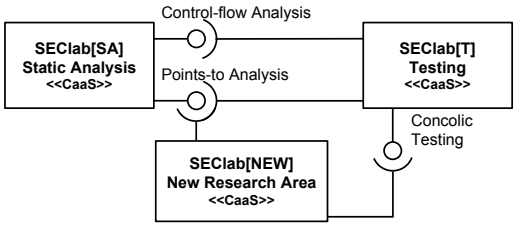

Figure 2: An example of new research built on top of CaaS.

(5) *System makes benchmark programs available for download.* Analysis Execution: Finally, SAClab makes benchmark programs (programs or their variants) available for download; if the researcher provided a Docker image, the system runs the benchmark programs as inputs to the analyzer in a container. To facilitate performance benchmarking, SAClab will provision a dedicated machine to run analyzers and capture CPU/memory/disk statistics.
(6),(7) *Researcher receives results and chooses dissemination options* The researcher downloads results with options to make data associated with their analysis tool public: the selection criteria, program transformation, benchmark programs, Docker image, output/results of the analysis, etc. Doing so enables researchers to share analysis tool details, benchmark results, and transformation techniques, leading to a faster pace of research and easily reproducible results.

Our example shows how a general functionality of SEClab naturally maps onto its SAClab instance. Moreover, SEClab can be instantiated for managing diversity in SE research [12]. In this instance the collaboratory obtains a definition of the program universe, dimensions, and similarity functions from the users; uses the universe definition to obtain suitable programs, and the curation component using the dimensions and similarity functions to obtain a diverse set of programs. Thus, we believe that the proposed vision of SEClab provides core necessary capabilities that satisfy the needs of the SE research community.

3 COLLABORATORIES AS A SERVICE (CAAS)

With establishing specific SEClabs and the advent of cloud computing, infrastructures as a service, etc, in the upcoming decades our vision could take us to a state where researchers will utilize established Collaboratories as a Service (CaaS). We envision a CaaS as a collaboratory that also operates as Software as a Service (SaaS). SaaS is a cloud-based system that provides applications, APIs, and other building blocks to facilitate easily building applications, for example as done with Dropbox, Salesforce, and Google Apps.

Many top SE and PL conferences have added an artifact evaluation process to the research tracks. Being optional, its results do not change the status of accepted papers (other than awarding an artifact badge). Some conferences, such as ECOOP, then publish the artifacts, helping ensure their availability to other researchers in the hopes of improving transparency and providing a basis for future research. Our vision is to help facilitate the adoption and use of *vetted* research artifacts by providing those artifacts as a software service available to the public at large.

Community Vetting of Provided Services. While there are many alternatives to structuring how research hosted on a collaboratory gets elevated to the status of being provided as a service, we propose that each research community utilizing the collaboratory develop their own guidelines. Such guidelines might include any of the following: require a peer-reviewed paper describing the research approach be published in certain conferences or journals; evaluations with given acceptance criteria; or meeting minimum performance guarantees for a provided benchmark.

Accelerating Research Using CaaS. Consider the diagram in Figure 2. In this figure, we show three different collaboratories: one for static analysis (SEClab[SA]), one for testing (SEClab[T]), and a new collaboratory for some future research area (SEClab[NEW]). In this figure, the first two also act as collaboratories as a service. Some examples of services provided might be computing control-flow, or advanced pointer analysis.

Notice the testing collaboratory makes use of some static analysis services, in addition to providing its own service for running concolic tests. The idea is to not just provide common functionality such as control-flow analysis as a library, but provide it as an ongoing service where the relevant research community can ensure the service is using the latest approved techniques. Thus, anyone using the service, such as the testing collaboratory, benefits from having the most state-of-the-art techniques available. They also can feel confident using them, knowing the SEClab[SA] has vetted (and continues to vet) the techniques.

Now when new researchers join the field, they have a plethora of available techniques to choose from. If this hypothetical new research area requires running concolic testing to identify particular inputs that cause a test suite to fail and then using pointer analysis to help localize the root cause, researchers only need to utilize available services, drastically increasing the speed of research.

4 CONCLUSION

In this paper we described our vision of software engineering research embracing the notion of shared collaboratories (SEClabs). The hope is SEClabs will help provide researchers the resources necessary to easily evaluate their own research. We also envision a future where research hosted at SEClabs is vetted by the community and made available as a Collaboratory as a Service (CaaS). This would further enable rapid advancement of research by providing community vetted resources for use in prototyping research ideas.

Undoubtedly our vision brings many challenges associated with its materialization and adaptation. Those challenges include converging on a set of common requirements for users of a specific collaboratory, organizing a self-sustainable collaboratory operation, and provide exceptional support for the users. With overcoming those challenges we believe researchers will have a great incentive to make use of and/or contribute to a SEClab, since it will provide them with a comprehensive state-of-the art experimental environment to perform empirical evaluations.

ACKNOWLEDGMENTS

This work supported by the US National Science Foundation under CNS-18-23357, CNS-18-23294, CCF-15-18776, and CNS-15-12947.

REFERENCES

[1] Stephen M. Blackburn, Robin Garner, Chris Hoffmann, Asjad M. Khang, Kathryn S. McKinley, Rotem Bentzur, Amer Diwan, Daniel Feinberg, Daniel Frampton, Samuel Z. Guyer, Martin Hirzel, Antony Hosking, Maria Jump, Han Lee, J. Eliot B. Moss, Aashish Phansalkar, Darko Stefanović, Thomas VanDrunen, Daniel von Dincklage, and Ben Wiedermann. 2006. The DaCapo Benchmarks: Java Benchmarking Development and Analysis. In *Proceedings of the 21st Annual ACM SIGPLAN Conference on Object-oriented Programming Systems, Languages, and Applications (OOPSLA '06)*. ACM, New York, NY, USA, 169–190. https://doi.org/10.1145/1167473.1167488

[2] Sylvia Dieckmann and Urs Hölzle. 1999. A study of the Allocation Behavior of the SPECjvm98 Java Benchmarks. In *Proceedings of the 13th European Conference on Object-Oriented Programming (ECOOP 1999)*. Springer Berlin Heidelberg, Berlin, Heidelberg, 92–115. https://doi.org/10.1007/3-540-48743-3_5

[3] Hyunsook Do, Sebastian Elbaum, and Gregg Rothermel. 2005. Supporting Controlled Experimentation with Testing Techniques: An Infrastructure and Its Potential Impact. *Empirical Softw. Engg.* 10, 4 (Oct. 2005), 405–435. https://doi.org/10.1007/s10664-005-3861-2

[4] Lisa Nguyen Quang Do, Michael Eichberg, and Eric Bodden. 2016. Toward an Automated Benchmark Management System. In *Proceedings of the 5th ACM SIGPLAN International Workshop on State Of the Art in Program Analysis (SOAP 2016)*. ACM, New York, NY, USA, 13–17. https://doi.org/10.1145/2931021.2931023

[5] Robert Dyer, Hoan Anh Nguyen, Hridesh Rajan, and Tien N. Nguyen. 2013. Boa: A Language and Infrastructure for Analyzing Ultra-Large-Scale Software Repositories. In *Proceedings of the 35th International Conference on Software Engineering (ICSE'13)*. 422–431.

[6] Robert Dyer, Hridesh Rajan, Hoan Anh Nguyen, and Tien N. Nguyen. 2014. Mining Billions of AST Nodes to Study Actual and Potential Usage of Java Language Features. In *Proceedings of the 36th International Conference on Software Engineering (ICSE 2014)*. ACM, New York, NY, USA, 779–790. https://doi.org/10.1145/2568225.2568295

[7] Docker Inc. 2018. Docker. https://www.docker.com/.

[8] Xuan-Bach D. Le, Duc-Hiep Chu, David Lo, Claire Le Goues, and Willem Visser. 2017. JFIX: Semantics-based Repair of Java Programs via Symbolic PathFinder. In *Proceedings of the 26th ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA 2017)*. ACM, New York, NY, USA, 376–379. https://doi.org/10.1145/3092703.3098225

[9] Xuan-Bach D. Le, Duc-Hiep Chu, David Lo, Claire Le Goues, and Willem Visser. 2017. S3: Syntax- and Semantic-guided Repair Synthesis via Programming by Examples. In *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering (ESEC/FSE 2017)*. ACM, New York, NY, USA, 593–604. https://doi.org/10.1145/3106237.3106309

[10] Pedro Martins, Rohan Achar, and Cristina V. Lopes. 2018. 50K-C: A Dataset of Compilable, and Compiled, Java Projects. In *Proceedings of the 15th International Conference on Mining Software Repositories (MSR'18)*. ACM, 1–5.

[11] Nuthan Munaiah, Steven Kroh, Craig Cabrey, and Meiyappan Nagappan. 2017. Curating GitHub for engineered software projects. *Empirical Software Engineering* 22, 6 (01 Dec 2017), 3219–3253. https://doi.org/10.1007/s10664-017-9512-6

[12] Meiyappan Nagappan, Thomas Zimmermann, and Christian Bird. 2013. Diversity in Software Engineering Research. In *Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering (ESEC/FSE 2013)*. ACM, New York, NY, USA, 466–476. https://doi.org/10.1145/2491411.2491415

[13] Suman Nakshatri, Maithri Hegde, and Sahithi Thandra. 2016. Analysis of Exception Handling Patterns in Java Projects: An Empirical Study. In *Proceedings of the 13th International Conference on Mining Software Repositories (MSR '16)*. ACM, New York, NY, USA, 500–503. https://doi.org/10.1145/2901739.2903499

[14] Michael Reif, Michael Eichberg, Ben Hermann, and Mira Mezini. 2017. Hermes: Assessment and Creation of Effective Test Corpora. In *Proceedings of the 6th ACM SIGPLAN International Workshop on State Of the Art in Program Analysis (SOAP 2017)*. ACM, New York, NY, USA, 43–48. https://doi.org/10.1145/3088515.3088523

[15] Koushik Sen, Darko Marinov, and Gul Agha. 2005. CUTE: A Concolic Unit Testing Engine for C. In *Proceedings of the 10th European Software Engineering Conference Held Jointly with 13th ACM SIGSOFT International Symposium on Foundations of Software Engineering (ESEC/FSE-13)*. ACM, New York, NY, USA, 263–272. https://doi.org/10.1145/1081706.1081750

[16] Vinayak Sinha, Alina Lazar, and Bonita Sharif. 2016. Analyzing Developer Sentiment in Commit Logs. In *Proceedings of the 13th International Conference on Mining Software Repositories (MSR '16)*. ACM, New York, NY, USA, 520–523. https://doi.org/10.1145/2901739.2903501

[17] Manu Sridharan and Rastislav Bodík. 2006. Refinement-based Context-sensitive Points-to Analysis for Java. In *Proceedings of the 27th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '06)*. ACM, New York, NY, USA, 387–400. https://doi.org/10.1145/1133981.1134027

[18] E. Tempero, C. Anslow, J. Dietrich, T. Han, J. Li, M. Lumpe, H. Melton, and J. Noble. 2010. The Qualitas Corpus: A Curated Collection of Java Code for Empirical Studies. In *2010 Asia Pacific Software Engineering Conference*. 336–345. https://doi.org/10.1109/APSEC.2010.46

[19] Christopher Vendome, Mario Linares-Vásquez, Gabriele Bavota, Massimiliano Di Penta, Daniel German, and Denys Poshyvanyk. 2015. License Usage and Changes: A Large-scale Study of Java Projects on GitHub. In *Proceedings of the 2015 IEEE 23rd International Conference on Program Comprehension (ICPC '15)*. IEEE Press, Piscataway, NJ, USA, 218–228. http://dl.acm.org/citation.cfm?id=2820282.2820314