Journal of Object Technology
Published by AITO — Association Internationale pour les Technologies Objets
http://www.jot.fm/

XCorpus – An executable Corpus of Java Programs

Jens Dietrich[a] Henrik Schole[b] Li Sui[c] Ewan Tempero[d]

a. Massey University, Palmerston North, New Zealand

b. Technical University of Dresden, Dresden, Germany

c. Massey University, Palmerston North, New Zealand

d. The University of Auckland, Auckland, New Zealand

Abstract Empirical studies on code require standardized datasets of significant size extracted from real-world programs in order to be reproducible and generalisable. We argue that there is a need for such data sets that are executable and can therefore be used for experiments using static and dynamic analysis. A harness for such a data set should have high coverage in order to facilitate the construction of comprehensive models of program execution.

We present XCorpus, a set of 76 executable, real-world Java programs, including a subset of 70 programs from the Qualitas Corpus. XCorpus uses a harness that is a combination of built-in and generated test cases, resulting in a branch coverage that is significantly better than what is available from DaCapo.

Keywords data set, benchmark, Java, empirical study, program analysis, test case generation, test coverage, dynamic program analysis

1 Introduction

Like all engineering disciplines, software engineering relies heavily on measurement. The advent of code repositories such as *SourceForge* [sou], *GitHub* [git] (for source code), and *Maven* [Mav] (for deployment artifacts) has given researchers in software engineering and programming languages access to large amounts of data to study. This allows them to gain a better understanding of the characteristics of real-world software, i.e., what it looks like and how measurements taken on code correlate to quality attributes.

Jens Dietrich, Henrik Schole, Li Sui, Ewan Tempero. *XCorpus – An executable Corpus of Java Programs*. Licensed under Attribution-NoDerivatives 4.0 International (CC BY-ND 4.0). In Journal of Object Technology, vol. 16, no. 4, 2017, pages 1:1–24. doi:10.5381/jot.2017.16.3.a1

Recently, there has been an increased emphasis on the reproducibility and generalisability of results obtained from empirical studies [CP16, Pen11]. In particular, reproducibility is quickly becoming an expectation to publish in certain areas, examples include the artifact evaluations which are part of ACM SIGPLAN conferences (including OOPSLA, POPL, PLDI and associated conferences like SAS, ECOOP and ESEC/FSE) [KV15]. SIGMOD [Boi16] and the biostatistics journal [Pen11] use a similar process, and there are efforts to standardise the artifact evaluation process [1]. It seems likely that this approach will be adapted by more conferences and journals in the future.

An important aspect to facilitate reproducibility as well as generalisability (i.e., to be able to generalise results to arbitrary programs of a certain class with reasonable confidence) and comparability (the ability to compare results with other research) is the use of standard data sets. The use of such data sets is common practice in many areas of computing, examples include the UCI machine learning repository [Lic13] and kaggle [kag]. In the context of studies on Java programs, two such data sets, DaCapo [BGH+06] and the Qualitas Corpus [TAD+10], have become widely used in the last 10 years.

We observe that data sets consisting of programs can be classified using the following categories according to their purpose:

1. **Code Corpus**: A corpus containing programs consisting of source and/or byte code and perhaps additional resources, organised in a way that facilitates static analysis. This can be achieved by using meta data and/or canonical file structures.

2. **Compilable**: Static datasets with additional scripts that support the compilation and building of datasets. This facilitates additional compile or build time analyses, and requires that the respective compile time program dependencies are resolved.

3. **Executable**: Compilable datasets with additional harnesses (drivers) to exercise the respective programs, facilitating dynamic analyses.

4. **Benchmarks**: Executable datasets with harnesses that facilitate performance studies. For instance, such harnesses include support for warmup runs and repeated executions.

The Qualitas Corpus is a code corpus, while DaCapo is a benchmark. Both datasets are curated, i.e. the programs used were actively selected to achieve certain aims related to variety, size and other aspects. What is more, some manual work was performed in order to facilitate experiments with those data sets. For instance, DaCapo provides a customisable harness to execute the respective programs. Its purpose is benchmarking, e.g., to compare the performance of different JVMs. The Qualitas Corpus provides a canonical structure for programs and meta data that facilitates measurements on code by means of static analysis.

The co-existence of several defacto standard data sets reflects their different focus on benchmarks (DaCapo) and static analysis (Qualitas Corpus). We argue that there are research questions where a dataset that combines the strength of DaCapo (being executable) and Qualitas Corpus (being larger, more diverse and up-to-date) is required. Use cases include studies that require a high level of soundness and precision

[1] `http://www.artifact-eval.org/` [accessed 20 March 17]

of the program analysis used and achieve this by offsetting the shortcoming of a particular analysis by combining it with a different analysis, or studies on program transformations that use models built by means of static analysis, but assess the impact of the transformation by executing programs before and after the transformation.

In this paper, we describe the construction of such a data set, the XCorpus. According to our classification used above, the XCorpus is an executable dataset, but not a benchmark. Use cases and requirements are discussed in detail in section 2. We then review related work in section 3, followed by a discussion of the various aspects of XCorpus in sections 4 (design) and 5 (implementation issues). We present the results of some measurements on the data set in section 6, and provide basic instructions how to obtain and use the XCorpus in section 7. A brief conclusion and discussion of future work can be found in section 8.

2 Use Cases and Requirements

We describe two scenarios for the type of research that motivates the construction of XCorpus. We then extract requirements for the data set from these use cases.

2.1 Assessing the Soundness and Precision of Static Program Analysis

Data sets such as the Qualitas Corpus consisting of large amounts of program code are particularly suitable for static analysis, including the measurement of properties by "looking at the code" without exercising it. While there is significant value in this, there are many questions static analysis alone cannot answer.

Static analysis is often assumed to be inherently *sound*, but with limited precision in order to maintain soundness [Ern03]. It has been noticed that in many real-world scenarios certain aspects of the program under analysis are not captured by static analysis and it is therefore also *unsound* [LSS+15]. This is often caused by the use of dynamic programming language features such as reflection, dynamic proxies, class loading, serialization and dynamic binding (`invokedynamic`). Those features are notoriously difficult to capture by static analysis techniques [LSS+15]. There are many analyses that are affected by this, including points-to and alias analysis, dependency-related metrics, call-graph construction, and automated recognition of design- and antipatterns. Programming language features that may cause unsoundness are seen as exotic by some. Nevertheless they are heavily utilised by higher-level frameworks, methods and patterns that are widely used in practice, such as dependency injection, service locators, aspect-oriented programming, middleware for distributed computing (CORBA, RMI, web services) and object-relational mapping frameworks (ORM).

Many static analyses are not precise either, as many algorithms trade-off precision for speed. This is particularly common for analyses that are based on high-complexity algorithms, such as the cubic bottleneck in points-to analysis [Rep98]. Precise context-sensitive points-to analysis is known to be NP-hard [Hor97]. Cross-referencing static analyses with dynamic analysis techniques where the program is executed and the respective program is observed is a possible approach to assess the level of (under– or over–) approximation. Examples are the observation of actual types of variables in order to assess the precision of call graphs in languages that use dynamic dispatch, and the accuracy of dependency-related metrics.

It is important to point out the limitation of the data set proposed here. Firstly, a user of the corpus presented here must still write additional scripts to gather data from

program execution, for instance, by injecting byte code that logs type information (instrumentation), or by gathering heap or thread dumps (sampling). Secondly, while cross-referencing static analysis with data from program execution can provide an *estimate* of the soundness and precision of the respective static analysis, it cannot be used to *accurately measure* it. We cannot expect that automated program execution will provide a "gold standard" against which soundness and precision of static analysis can be assessed, as we cannot guarantee that the execution of the program will visit all possible, or even just all meaningful (from the point of view of the user) states.

However, the use of a corpus of real-world programs is useful to assess the performance of static analysis tools on real-world programs.

2.2 Correctness of Program Transformations

The second use case for an executable corpus is to use it in order to build models of program correctness. While there is a large body of research on formal models of correctness and how to verify programs against them by means of static analysis, these approaches are not always useful to assess real-world programs as these models do not exist for these programs, and the creation of these models is often not feasible. However, some notion of correctness is necessary to demonstrate the impact changes to a program have on its behaviour. Examples where this is required include changes to the compiler or the runtime, and the (automated) refactoring of code. Having an executable corpus will allow researchers to demonstrate (although not formally prove) limited correctness using the same approach used in industry: re-compile and re-execute the code after the changes made, and observe its behaviour, preferably by means of regression tests with assertions describing valid program states. Therefore, an executable corpus can provide an approximated model of correctness. A variant of this use case is to assess the impact of transformation on non-functional quality attributes, such as performance.

Again, it is important to understand the limitation of the corpus here. While there are programs with good test suites that describe the semantics of the program well, this seems to be the exception. Therefore we decided to complement existing program entry points by generated tests. Test generation is driven by coverage goals and the algorithms to generate tests are optimised for this goal. But tests also describe program semantics through their output: constraints that express the state of the program after the execution of tests. This is usually done through assertions[2]. Generating concise assertions is an unsolved problem, known as the *Oracle problem* in the testing community [MH81]. There is a large body of research on oracle generation [BHM+15]. While generated oracles are valuable, we cannot expect that they precisely capture the semantics of the program as intended by the programmer.

2.3 Requirements

We start the presentation of the corpus with a discussion of requirements. We adopted the first two requirements directly from DaCapo. For requirement 2, we add customisation.

[2] Assertions in tests are used to express postconditions and invariants. Some testing frameworks including *junit* also provide an explicit syntax for preconditions via assumptions, although this feature is not widely used in our experience.

REQ1 **Diverse real-world applications**. Real-world here means in particular *non-synthetic* programs, i.e. the programs were not just created for the purpose of being used in this data set.

REQ2 **Ease of use**. We want the applications to be relatively easy to use, measure and customise.

REQ3 **High coverage**. This is to provide enough information to build comprehensive and detailed models of program execution as motivated by the use cases discussed above.

REQ4 **Presence of Assertions**. This is to provide enough information to build comprehensive and detailed models of program execution as motivated by use case two.

REQ5 **Currentness**. We want the corpus to be fairly up-to-date. In particular, it should contain programs using modern language features, which might cause some of the issues discussed in Section 2.1.

REQ6 **Large, yet manageable**. We want the corpus to be of a significant size so that results obtained are generalisable – this requires size and diversity. However, it should still be manageable, both in terms of size and time needed to process it, i.e. to exercise all programs. Size is a necessary, but not a sufficient condition for representativeness. A larger, more diverse corpus will be more representative than a subset it contains, so representativeness and generalisability are relative, not absolute concepts.

REQ7 **Extensibility**. We want to provide a data set that is easy to extend. In particular, the scripts used to setup and exercise the programs in the data set should work for additional programs that can be "dropped-in" by a users. This facilitates a *live data set* that allows users to create and share extensions, taking advantage of the infrastructure provided by XCorpus.

3 Related Work

Empirical studies on code from real-world programs started in the seventies, notable examples include the work of Knuth on 400 FORTRAN programs [Knu71] and Chevance and Heidet on 50 COBOL programs [CH78]. For the reminder of this section, we focus on data sets and benchmarks containing Java programs.

DaCapo [BGH+06] is a small benchmark that had two major releases, 2006 and 2009. DaCapo 2009 has 14 real-world programs, including a customisable test harness to exercise those programs. DaCapo 2006 contains only 11 programs. The small size of the data set makes it difficult to generalise results obtained with it. An example for this is the study on novel high-performance algorithms for points-to analysis by Dietrich et al [DHS15]. Experiments on DaCapo 2009 show that that a top-down refinement strategy to compute the fix point of the iterative points-to algorithm has a very high precision. However, the same experiment conducted on *bloat* (part of DaCapo 2006, but removed from DaCapo 2009) yields a very different result as precision of the same algorithm is as low as 40 %. This example illustrates that experiments on small data sets often miss important aspects that are present in real world programs.

The **Qualitas Corpus** [TAD+10] is a large collection of real-world Java programs. The first publicly available version was published in 2008, and frequently updated after this. The last edition at the time of writing this was published in 2013 (version 20130901). This edition consists of 754 versions of 112 programs. The Qualitas Corpus does not provide a harness.

Qualitas.class [TMVB13] is an extension of the Qualitas Corpus version 20120401, making the respective projects compilable. In particular, library dependencies are resolved. Qualitas.class does not provide a harness, it is still mainly used for static analysis.

Spec [spe] has published several Java benchmarks, in particular **SPECjvm2008** [SCWP09]. SPECjvm2008 contains a combination of several executable synthetic data sets and incorporates several real-world programs such as *sunflow* and *derby*. The focus of the benchmark is on CPU-intensive tasks such as compression, encryption and mathematical calculations such as matrix multiplication and floating point operations. SPECjvm2008 also incorporates the **SciMark 2.0** benchmark [PM00].

The **Software-artifact Infrastructure Repository (SIR)** [DER05] is a widely used data set consisting of a combination of synthetic and real-world programs. SIR contains programs implemented in different languages, 68 of them are Java programs, and in some cases different versions of the same program are part of SIR. Some (synthetic) programs are tiny (for instance, *wronglock* has only 38 LOC). SIR's focus is on testing, i.e., there are meta-data to expose test cases and fault data (both real and seeded).

The **Purdue Benchmark Suite (PBS)** by Grothoff et al. [GPV01] was created and used in a study of confined types. It is relatively small, consisting of 33 Java systems of which 5 have more than 200 classes, and a total of 46,165 classes. It was influential in the design of the Qualitas Corpus.

Table 1 summarised related data sets and benchmarks.

Table 1 – Overview of Java datasets and benchmarks (ver. - whether the data set contains multiple versions of some programs, synth / r.w. - whether the program contains synthetic / real-world programs).

name	year	updated	programs	harness	ver	synth	rw
DaCapo	2006	2009	14	yes	no	no	yes
Qualitas Corpus	2008	2013	112	no	yes	no	yes
qualitas.class	2013	-	111	no	no	yes	yes
Sir	2005	-	68	yes	yes	yes	yes
SPECjvm2008	2008	2008	10	no	yes	yes	yes
PBS	2001	-	33	no	no	no	yes

In recent years, numerous studies have used open, un-curated repositories such as Maven, GitHub and GoogleCode. Examples include the work of Raemakers et al. on evolution and semantic versioning using data from Maven [RvDV16], Lopes's and Ossher's work on how scale affects the structure of Java programs [LO15] using projects hosted on GoogleCode, and the work by McDonnell et al. on API stability of Android applications using projects from GitHub [MRK13].

To facilitate the work with various large repositories, Bajracharya et al. have proposed **sourcerer** [BOL14], an infrastructure for analysis and search of various online repositories. Sourcerer contains crawler plugins to collect projects from several popular open source repositories.

4 The Design of XCorpus

This section discusses the design decisions we made, driven by the requirements presented in Section 2.3. Requirements such as ease of use and manageable size are highly subjective. Therefore, we do not claim that we meet those requirements, this is for future users to decide.

4.1 Diverse Real-World Applications

The core of the XCorpus is a subset of 70 programs from the Qualitas Corpus version 20130901. The criteria used to include programs in the corpus are discussed in [TAD$^+$10], and ensure that a wide range of diverse real-world applications are used.

However, the XCorpus does not contain all programs from the corpus for a variety of reasons.

1. **Blacklisted classes:** Classes and entire packages are blacklisted by the *evosuite* tool we used for test case generation[3]. Example of corpus programs affected by this are *cobertura* and *myfaces_ core*.
2. **Unresolvable dependencies:** We used the Maven repository to resolve dependencies. In some cases, transitive dependency resolution failed. This was the case for *springframework* and *jasperreport*.
3. **Failing built-in tests**. We excluded *roller* as all tests resulted in an unrecoverable error ("Forked Java VM exited abnormally").
4. **Environment configuration**. An example is *hsqlda* which requires a sql server connection to be exercised.
5. **Timeouts**. As discussed below in Section 5.3, the XCorpus utilises automated test case generation. This is very resource intensive, and in some cases generation timed out after one week. An example is *heritrix* and we decided to exclude those programs.
6. **Test case generation failure**. In one case (*xmojo*) test case generation failed. As this program did not have any built-in tests either, we removed this from the corpus.

4.2 Ease of Use

The corpus uses a canonical internal structure described in section 4.10 that facilitates many kinds of analysis.

For the harness, we use a different approach to DaCapo, which uses a Java executable. We provide standard *ant* [ant] build scripts with dedicated targets to exercises the respective programs. There are project-specific local scripts to build and exercise individual programs as well as a global script to exercise all programs. The *ant* scripts can be customised to enable pre- and postprocessing, facilitating tasks like instrumentation.

[3] `https://github.com/EvoSuite/evosuite/blob/master/runtime/src/main/resources/excluded.classes` [accessed 20 March 17]

4.3 High Coverage

Generally, the coverage provided by built-in executables and test cases is very low, and often there are no built-in tests or executables. We address this problem by augmenting built-in tests with generated tests. This is sufficient to attain a better coverage. Details are reported below in Section 6.

4.4 Presence of Assertions

While many programs contain assertions, either in the form of Java assert statements, assertion checks in built-in test cases or some other contractual mechanisms such as contract APIs, test case generation can be used to provide more assertions. As discussed in Section 2.2, those have to be used with caution.

4.5 Currentness

The XCorpus is derived from the Qualitas Corpus version 20130901. This was the latest version available when the project started, and compares favourable with DaCapo, last updated in 2009. XCorpus also contains some additional programs with versions released in 2016/17. Appendix B provides an overview of language features used by programs in the corpus, it clearly demonstrates the use of modern language features such as annotations, generics and lambdas.

4.6 Large, yet manageable

The XCorpus contains 76 programs, the total size of the distribution is 633.2 MB (735.5 MB uncompressed). The size of the distribution is greatly reduced by using declarative dependencies. As the harness consists of *ant* scripts, *ant* is also used to build the respective projects. We manually created the respective scripts and represent library dependencies by symbolic dependencies pointing to the Maven repository. This is achieved by using *ivy*[4]. The dependencies are transparently resolved during build before the programs are exercised.

We were able to run all generated tests on commodity hardware. We tested on a Ubuntu 16.04 LTS with a 2.7 GHz Intel Core 7 Processor, using a Java(TM) SE Runtime Environment (build 1.8.0_111). Executing all generated tests took 20 hours, 21 mins, and executing all built-in tests took 1 hour 33 min.

4.7 Extensibility

While it is desirable to have a stable core of programs to facilitate comparable and reproducible research, it is equally important to have the flexibility to extend the data set in order to add programs that have particular features or characteristics. The XCorpus supports this through *extension sets*. Extension sets are programs stored in a canonical data structure with a root folder named *xcorpus-extension-<id>*. The XCorpus then offers support to work with these extensions, for instance, scripts to run tests across the core data set and all present extensions.

[4] `ant.apache.org/ivy/`

4.8 The Standard Extension

At the time of writing, XCorpus contains one extension *xcorpus-extension-20170313*, we refer to it here as the *standard extension*. It consists of six programs, selected based on popularity and the use of interesting dynamic language features that are of particular interest for researchers interested in the differences between static and dynamic analysis. An overview of the features used by programs in XCorpus in general and in the standard extension in particular is provided in Appendix B.

1. *guava-21.0* is a utility library that uses many modern language features and system APIs, including lambdas / `invokedynamic` , `sun.misc.Unsafe` , weak and soft references
2. *mockito-all-1.10.19* is a testing framework that makes comprehensive use of dynamic proxies
3. *drools-7.0.0.Beta6* is a rule engine that compiles code at runtime using the JSR199 compiler API [jsra]
4. *asm-5.2* is a byte code analysis and manipulation framework
5. *jmeter_core-3.1* is a stress test framework that uses an embedded language through the JSR223 scripting API [jsrb]
6. *jasperreports-1.1.0* is a reporting framework, we use a version compiled with the dynamo compiler [JD16], this will create `invokedynamic` instructions not related to lambdas and therefore not using `LambdaMetaFactory` for bootstrapping

4.9 Limitations

XCorpus does not explicitly support multiple runs of the same harness and JVM warmup runs. It is therefore not suitable to run performance benchmarks.

4.10 The Structure of the XCorpus

Figure 1 shows the file structure of the XCorpus distribution. The `/tools` folder contains the global scripts and several utility classes, in particular the assertion adapters that will be discussed in section 5.4 in more detail. The global exercise script `/tools/build.xml` has several targets to process all programs.

The `/data/qualitas_corpus_20130901` contains the binaries and required resources of the actual corpus projects. Each project has its own folder, such as `/data/qualitas_corpus_20130901/antlr-3.4`. Project specific scripts and generated tests are located in folders `<project>/.xcorpus`. The source code of the generated tests is included. The respective source code folders are compressed in a file `evosuite-tests.zip`.

Some corpus projects contain several binaries (jar files). An example is *aspectj-1.6.9* with several binaries for its tools, runtime and weaver components. In these cases we decided to keep the structure and create a layer of virtual *subprojects* that were processed separately.

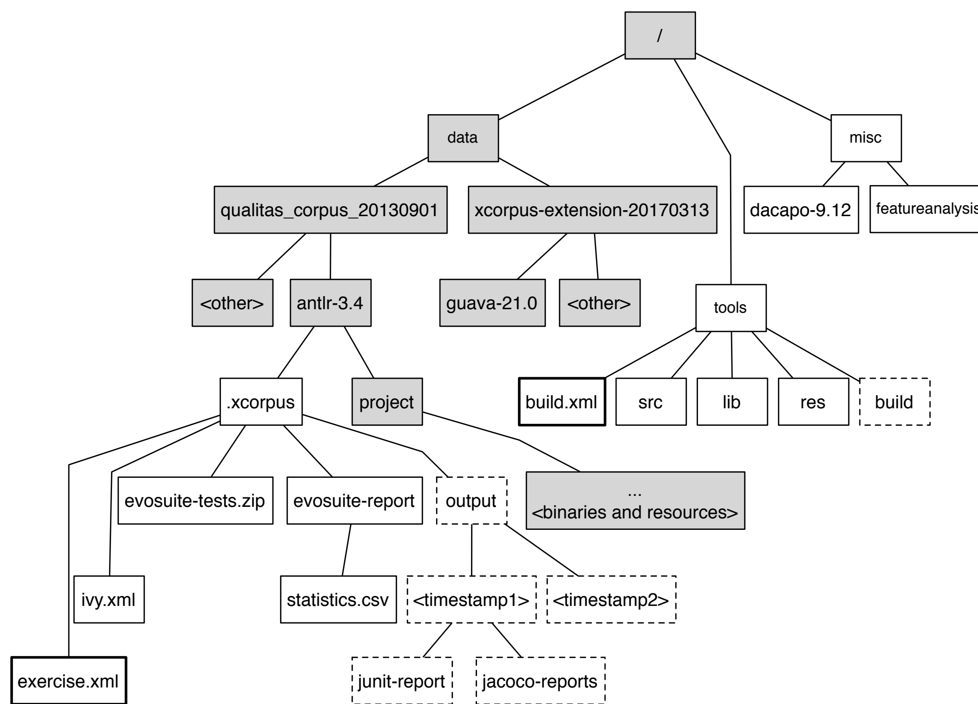


Figure 1 – XCorpus folder structure (simplified). Exercise scripts are highlighted with bold borders. Folders represented by boxes with dashed lines are generated during the execution of exercise scripts, and are not part of the distribution.

5 Implementation Issues

5.1 Scripts and Dependency Management

The exercise scripts are implemented as *ant* scripts. This ensures platform independence, and strikes a good balance between comprehensibility and the flexibility to customise and modularise scripts in particular when compared with similar tools such as *maven* and *gradle*.

One particular design objective was to keep the size of the distribution manageable (REQ6). In particular, the XCorpus is hosted on `bitbucket.com`, which imposes an overall repository size limit. The distribution size was first reduced by making the decision to not include source code, as pre-compiled binaries are available as part of respective Qualitas Corpus distribution. Furthermore, libraries XCorpus programs depend on are not distributed in most cases either, as most libraries can be found in the Maven repository. We used *ivy* [ivy], an *ant* extension for automated dependency management, in order to transparently resolve (potentially transitive) library dependencies. We always reference specific versions instead of version ranges in order to ensure reproducibility of results[5].

The XCorpus does not support the compilation of programs from sources. This has some impact on the usability of the corpus, in particular for the kind of application discussed in Section 2.2 where the impact of *source code* transformations is to be assessed. A user interested in such experiments would have to provide scripts to compile programs, for instance, by integrating XCorpus with the Qualitas.class project [TMVB13] that provides a compilable version of the Qualitas Corpus, or by integrating project-specific build scripts.

5.2 Existing Program Entry Points

There are several ways to exercise a program. Firstly, there are a few programs that do have `main` methods that can be directly executed without supplying runtime parameters or input files. Examples include *jgraph* and *jrat*. Often, while invoking these `main` methods will start a program, the program does not always execute automatically in a meaningful way as it runs in an interactive mode where user input is required. Typical examples are programs that use a graphical user interface, such as *jrat*. While it is possible to simulate the events caused by user interaction, the setup is tedious and has the side effect that XCorpus-based experiments must be conducted on a machine that supports user interfaces[6]. Therefore, we decided against the use of `main` methods as entry points to exercise the program.

An alternative approach is to use test cases. The use of unit / regression tests is widely considered as best practice by Java programmers, and therefore many of the corpus programs have built-in unit tests. If this was the case, we integrated these tests into the execution scripts. We looked for *junit* test cases, and found 33 programs with at least one `junit` test. For those programs, we found and integrated 31,906 test cases (*junit* methods, counting methods in parametrised tests as one) organised in

[5] The use of version ranges is based on the idea of semantic versioning [sem] which stipulates that certain version changes imply compatibility. However, several empirical studies have demonstrated that developers do not adhere to the principles of semantic versioning and incompatible API changes are common even if only the micro version of a program is changed [DJB14, RvDV16].

[6] For instance, a Linux machine would have to provide an *xserver*.

4,536 classes [7].

We are aware of at least one program (*tapestry*) that uses the alternative *testNG* testing framework. We did not integrate these tests.

However, there is still a large number of programs without built-in tests, and even if the program has tests, test coverage is often low, indicating that large parts of the respective program are not exercised by tests.

5.3 Generated Tests

For this reason, we complemented built-in tests by generated tests using the *evosuite 1.0.3* test generation framework [FA11]. *Evosuite* uses a search-based approach to generate test suites. Test generation is guided by coverage criteria that can also be customised [RCV+15]. By default, branch coverage is used as a criterion. It has been demonstrated that *evosuite* can generate tests with high branch coverage [FA16]. There are multiple alternative coverage criteria supported by *evosuite*, including line, method and top-level method coverage.

Test case generation with *evosuite* is highly customisable and new versions are published frequently. Therefore, we provide generated test suites as part of the distribution but acknowledge that some users might want to regenerate the tests with settings that suit their particular needs. For this reason, the XCorpus distribution also contains scripts that can be used to re-generated all tests. This is discussed further in section 5.6.

We generated a total of 295,843 tests (*junit* methods) organised in 62,574 classes. The respective source code can be found in the repository in the `<project>/.xcorpus/` folders.

5.4 Dealing with Assertions

Evosuite can also generate assertions. Tests with generated assertions sometimes fail during the execution of generated tests, for instance, due to non-deterministic program execution. The XCorpus provides a built-in mechanism to disable assertions in generated tests by using the `AssertionAdapter`. The adapter uses standard JUnit assertions if assertion checking is enabled, and ignores assertions otherwise. The respective setting is an entry in `/tools/res/commons.properties`, and is used by the global exercise script as well as by the local, project-specific scripts.

5.5 Dealing with Exceptions and Errors

The execution of Java programs can fail for many reasons. This results in exceptions being thrown, usually runtime exceptions, indicating that the program has reached an illegal state. Programs may also fail with checked exceptions, for instance if `main` is declared with a `throws Exception` clause, or with errors, indicating problems in the JVM, usually related to memory allocation (insufficient heap or stack space) or linking.

In order to build some fault tolerance into the exercise scripts, unit tests are executed with the *ant* `junit` task with the `fork` flag set to `true`, and `forkMode` using the default `perTest` value. This means that each tests is executed in a separate JVM instance, and exceptions and errors during test execution will not prevent other tests

[7] The respective script to count builit-in and generated tests is available from the `/misc/featureanalysis` folder in the XCorpus repository.

from being executed. Two parameters can be configured to set the timeout of forked JVMs, and the amount of heap memory forked JVMs can use.

A drawback of this design is that some operating systems with a graphical user interface, in particular OSX, detect problems in crashed JVM instances and try to notify the user with modal warning dialogues, therefore interrupting the execution of the script. It is recommended to disable those OS functions, or to run experiments on a headless server operating system.

5.6 Regenerating Test Cases

Users may want to re-generate tests in order to obtain better coverage. This could be done either by (1) using an improved updated version of *evosuite*, (2) changing the coverage criteria in the *evosuite* settings or (3) by granting more resources (time and memory) to *evosuite* to generate tests.

To facilitate this, we added a `generate-tests` target to the `exercise.xml` script. This is very expensive to run, and based on the configuration and the hardware used, this script could take *multiple weeks* to complete.

However, this task can easily be parallelised by generating tests separately per program.

6 Coverage Data

One of the main requirements for the XCorpus was to obtain a harness with high coverage. To this end, we measured branch coverage for three data sets.

First, we used the *jacoco* coverage tool [jac] to measure branch coverage for the programs in DaCapo 2009, exercised with the provided driver using the default workload [8]. The average coverage is 16.10%. Details are shown in figure 2. This figure shows the projects ordered by coverage with their respective branch coverage value. The programs with the highest coverage are *tradebeans* (25.82%) and *tradesoap* (21.93%) [9], followed by *sunflow* (21.59%), the programs with the lowest coverage are *xalan* (7.65%), *jython* (9.94%) and *batik* (10.05%).

Next, we measured the branch coverage of the 28 XCorpus programs with built-in *junit* tests, again using *jacoco*. The average coverage is 34.42%, details are shown in figure 3. The programs with the highest coverage are *jFin_DateMath-R1.0.1* (77.60%), *commons-collections-3.2.1* (75.72%) and *checkstyle-5.1* (72.10%), whereas *freecol-0.10.7* (1.26%), *wct-1.5.2* (1.66%) and *findbugs-1.3.9* (3.96%) have all a very low coverage.

We investigated the branch coverage of generated test cases using the evosuite-generated branch coverage reports when test cases are generated. The result is shown in figure 4.The average *evosuite* branch coverage for the 70 projects is 55.86%. Projects with high coverage include *jFin_DateMath-R1.0.1* (93.33%), *emma-2.0.5312* (87.63%) and *commons-collections-3.2.1* (85.57%), the programs with the lowest coverage are

[8] Using the large workload does not generally lead to higher coverage, and the larger workload is not available for all programs. These issues are discussed in [BSSM10] in more detail.

[9] The two *daytrader* benchmarks are described in `http://dacapobench.org/daytrader.html` [accessed 30 March 17]. Both *tradebeans* and *tradesoap* use the same server workload but interact with the server using a web client via soap (*tradesoap*) or a direct api (*tradebeans*), respectively. The use of different client packages explains the difference in the coverage values measured. There is also a small difference in coverage for the *daytrader* core package `org.apache.geronimo.samples.daytrader` caused by the precense of an additional method `TradeAction#runDaCapoTrade` in *tradesoap*.

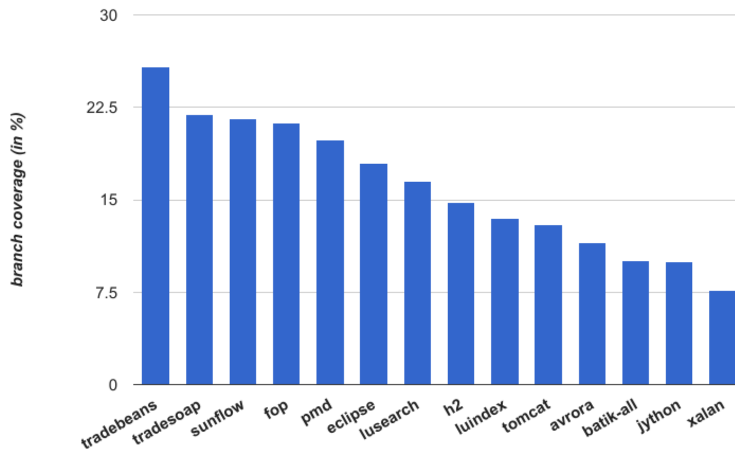


Figure 2 – Branch coverage of DaCapo programs exercised with the DaCapo harness with the default workload (as reported by jacoco)

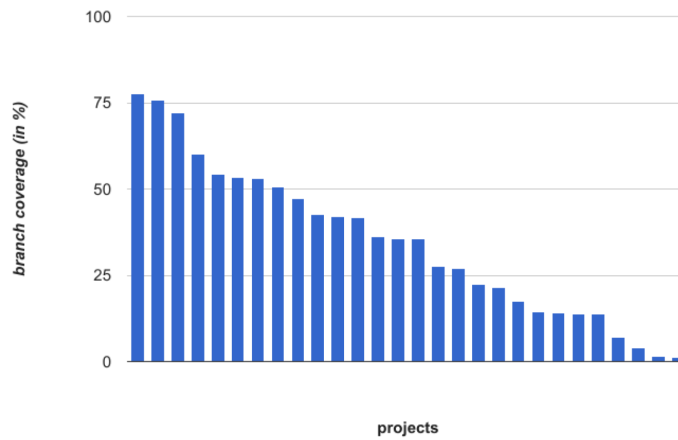


Figure 3 – Branch coverage of programs from the XCorpus taken from the qualitas corpus, exercised using built-in tests (as reported by jacoco)

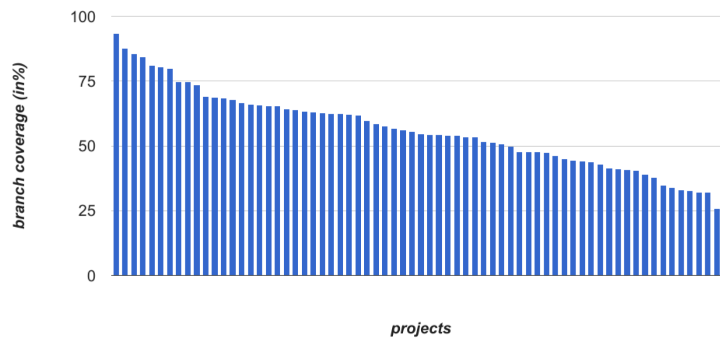


Figure 4 – Branch coverage of programs from the XCorpus taken from the qualitas corpus, exercised using generated tests (as reported by evosuite)

freecs-1.3.20100406 (20.87%), *megamek-0.35.18* (25.75%) and *jparse-0.96* (32.08%). These results are consistent with the results *evosuite* has achieved in the recent SBST competition, reported in [FA16]. These results clearly show that generated test cases yield high coverage. This should allow building models that can provide a good approximation of all possible program behaviours.

Finally, we measured the coverage for the projects in the standard extension separately. The average coverage is high with 62.35 % for built-in and 60.25 % for generated tests. Details are shown in figures 5 and 6, respectively. We do not include built-in tests for *jasperreports* because we use a version that was compiled with dynamo compiler [JD16].

7 Obtaining and Using the XCorpus

7.1 Obtaining the Corpus

The XCorpus can be cloned from a mercurial repository using the following command:

```
hg clone https://jensdietrich@bitbucket.org/jensdietrich/xcorpus
```

On the project web site, the current version can be found. To obtain a certain version, the corpus should be cloned using the `-u` option, such as:

```
hg clone https://bitbucket.org/jensdietrich/xcorpus -u 1.0.0
```

This will create the file structure shown in figure 1 on the local hard drive. We plan to actively maintain and update the corpus and release new versions, an explicit (semantic) versioning policy can be found in Appendix C.

7.2 Using the Corpus

7.2.1 Stability of Execution Environment

Given the large number of tests being executed, the actual test results might differ between execution platforms and potentially even between execution runs on the same

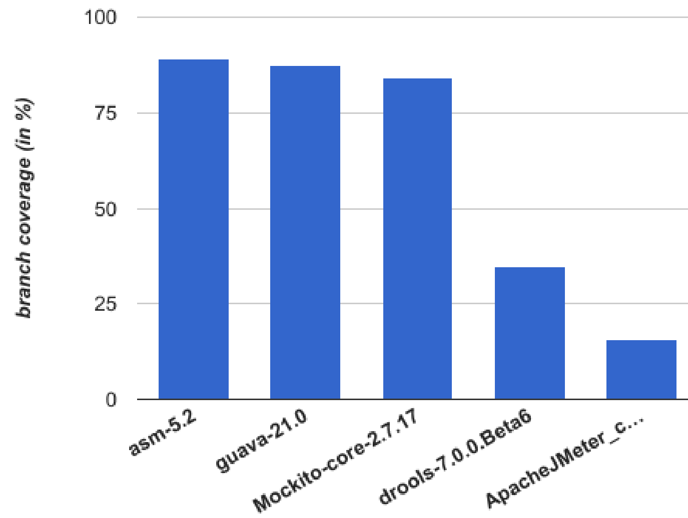


Figure 5 – Branch coverage of programs from the standard extension, exercised using built-in tests (as reported by jacoco)

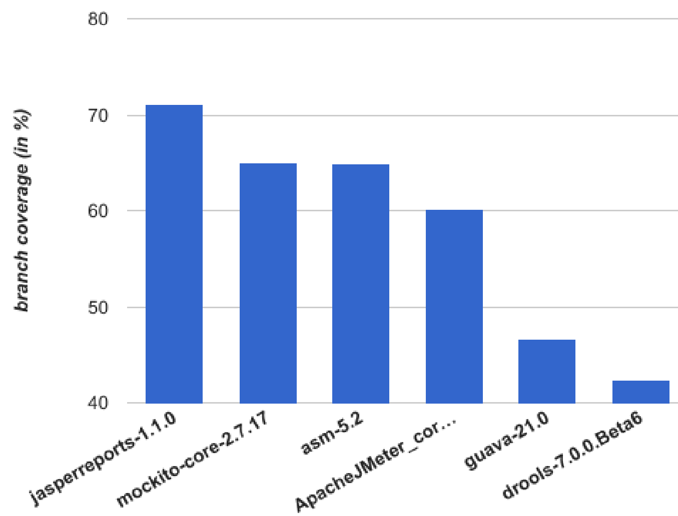


Figure 6 – Branch coverage of programs from the standard extension, exercised using generated tests (as reported by evosuite)

platform. While this might sound unsatisfactory, we believe it is unavoidable given the size and the complexity of the corpus. Some of the programs have dependencies to certain operation systems and configurations (e.g., whether a *xserver* is available), some long running tests might time out on some slower system, but succeed on faster systems, and some programs have non-deterministic features.

To exercise the XCorpus, two aspects are desirable from the users point of view: *speed* and *stability of results*. We describe two methods how to use the XCorpus, each representing a different trade-off between those two qualities.

7.2.2 Native Execution

To use the corpus directly using Java on the host operating system, the following software must be installed :

1. Java 8 must be installed and the *java* and *javac* executables must be in the path
2. *ant-1.9.7* or better, `ANT_HOME` set to point to the *ant* installation root folder, and the *ant* executable must be in the path
3. *ivy-2.4.0*, *junit-4.12.jar* and *hamcrest-1.3.jar* must be installed by copying the jar into `$ANT_HOME/lib`

To run the tests for a particular program, execute *ant* with the respective project-specific build script `<project>/.xcorpus/exercise.xml`. There are three relevant targets to run built-in tests (`builtin-tests`), generated tests (`generated-tests`) or all tests (`all-tests`). For instance, the following command runs all built-in tests.

```
ant -f exercise.xml builtin-tests
```

Global scripts that can be used to exercise all programs are located in `tools/-build.xml`. In particular, the `run-all-test` target exercises all tests in all programs. The build scripts also contains targets to regenerate the *evosuite* tests. Test case generation and execution can be customised in `tools/res/commons.properties`. In particular, assertion checking can be switched on by editing this file, and the timeouts and maximum heap size for the forked JVMs used to execute tests can be set.

The `README.md` file included in the distribution and the various wiki pages (`https://bitbucket.org/jensdietrich/xcorpus/wiki/Home`) provide further information.

7.2.3 Execution in a Container

The distribution contains a *Dockerfile* to build a light-weight, linux-based container. The detailed instructions are in the project *README.md* file. Using the docker container has the advantage that a more stable and repeatable environment is provided. In particular, fixed versions of the *OpenJDK*, *ant* and the various libraries required are used.

8 Conclusion and Future Work

In this paper, we have introduced XCorpus, a data set suitable for both static and dynamic analysis that addresses the shortcomings of several other data sets widely used in studies on Java code.

An obvious limitation of XCorpus is that it will quickly age, and new versions have to be compiled frequently in order to keep it relevant. At the moment, this process is

very expensive due to (1) the manual work required to build the projects and (2) the computing resources required to generate the test cases. We expect that both aspects can be addressed in the near future as many projects move to formats with a canonical project structure and rich meta data (addressing (1)), and test case generation can be automated using cloud computing.

9 Acknowledgement

This project was supported by a gift from Oracle Labs Australia to the first author.

A List of Programs in XCorpus

Table 2 – List of XCorpus programs

Programs from Qualitas Corpus 20130901		
aoi-2.8.1	jasml-0.10	mvnforum-1.2.2-ga
aspectj-1.6.9	javacc-5.0	nekohtml-1.9.14
axion-1.0-M2	jedit-4.3.2	openjms-0.7.7-beta-1
batik-1.7	jena-2.6.3	oscache-2.4.1
c_jdbc-2.0.2	jext-5.0	picocontainer-2.10.2
castor-1.3.1	jFin_DateMath-R1.0.1	pmd-4.2.5
checkstyle-5.1	jfreechart-1.0.13	pooka-3.0-080505
colt-1.2.0	jgraph-5.13.0.0	proguard-4.5.1
commons-collections-3.2.1	jgraphpad-5.10.0.2	quartz-1.8.3
displaytag-1.2	jgrapht-0.8.1	quilt-0.6-a-5
drawswf-1.2.9	jgroups-2.10.0	sablecc-3.2
emma-2.0.5312	jhotdraw-7.5.1	sandmark-3.4
findbugs-1.3.9	jmoney-0.4.4	squirrel_sql-3.1.2
fitjava-1.1	jparse-0.96	sunflow-0.07.2
fitlibraryforfitnesse-20100806	jpf-1.5.1	tapestry-5.1.0.5
freecs-1.3.20100406	jrat-0.6	tomcat-7.0.2
galleon-2.3.0	jrefactory-2.9.19	trove-2.1.0
htmlunit-2.8	jspwiki-2.8.4	velocity-1.6.4
informa-0.7.0-alpha2	jsXe-04_beta	wct-1.5.2
itext-5.0.3	log4j-1.2.16	webmail-0.7.10
Ivatagroupware-0.11.3	lucene-4.3.0	weka-3-7-9
jag-6.1	marauroa-3.8.1	xalan-2.7.1
james-2.2.0	megamek-0.35.18	xerces-2.10.0
freecol-0.10.7 [10]		
Programs from *xcorpus-extension-20170313*		
guava-21.0	mockito-all-1.10.19	drools-7.0.0.Beta6
asm-5.2	jmeter_core-3.1	jasperreports-1.1.0

Table 2 lists the names and versions of the programs contained in the XCorpus.

B Language Features used by Corpus Programs

Table 3 provides an overview of the features present in XCorpus programs. These numbers were obtained by means of byte code analysis [11]. We analysed properties of implemented artifacts (superclasses, interfaces, method flags, ..), and call sites of certain methods. This includes allocation sites represented as call sites of `<init>`. We distinguish between programs from the Qualitas Corpus and programs from the extension, and report the total number of programs in which a feature occurs, and the average of occurrences for all programs that contain the respective feature at least once.

The features were selected taking three aspects into account: (1) the presence of byte code that make feature detection possible (for instance, the use of auto-boxing or annotations with a `SOURCE` retention policy cannot be detected in byte code) (2) the currentness of a features (for instance, the use of lambdas / invokedynamic is of interest as lambdas were only introduced in Java version 8) (3) whether a feature is relevant for studies related to one of the motivating use cases (for instance, the use of reflection is of interest as it often hampers the soundness of static analysis).

This table is no replacements for a systematic study on "how language feature X is used by real-world Java programs", it is intended to illustrate the feature richness of the programs in the XCorpus, and to inform the user on whether it is suitable for a certain purpose.

The source code of the scripts used for feature extraction is available from the XCorpus repository, and can easily be customised and extended to extract other features.

[11] The respective scripts are available from the `/misc/featureanalysis` folder in the XCorpus repository, these scripts also produce (rather large) CSV files with details about how features are used by particular programs

Table 3 – Features in programs included in the XCorpus (c.s.- call site, $j - java, $l - lang, $u - util, $r - reflect, $jx - javax, # - method or constructor in, - wildcard)

		qualit. corp.		std. extension	
group	feature	count	avg	count	avg
proxies	impl. of $j.$l.$r.InvocationHandler	10	1.40	3	1.67
proxies	c.s. of $j.$l.$r.Proxy#newProxyInstance	10	1.60	3	1.33
reflection	c.s. of $j.$l.$r.Method#invoke	47	15.60	5	9.00
reflection	c.s. of $j.$l.$r.Constructor#newInstance	37	5.05	5	6.60
reflection	c.s. of $j.$l.$r.Field#get*	29	3.83	4	6.25
reflection	c.s. of $j.$l.$r.Field#set*	12	3.75	3	3.00
reflection	c.s. of $j.$l.Class#newInstance	58	11.50	4	20.75
reflection	c.s. of $j.beans.Introspector#*	13	3.62	2	4.00
reflection	c.s. of $j.$u.ServiceLoader#*	0	n/a	1	2.00
reflection	c.s. of $j.io.ObjectInputStream#*	39	22.79	5	26.40
reflection	c.s. of $j.beans.XMLDecoder#*	5	4.60	0	n/a
classloading	c.s. of $j.$l.ClassLoader#*	51	19.51	6	21.00
classloading	c.s. of $j.security.SecureClassLoader#*	2	2.50	0	n/a
classloading	c.s. of $j.net.URLClassLoader#*	20	4.80	1	2.00
classloading	c.s. of $j.rmi.server.RMIClassLoader#*	0	n/a	0	n/a
invokedyn.	subtype of any type in $j.$u.function	0	n/a	1	12.00
invokedyn.	c.s. of $j.$u.function.*#*	0	n/a	2	59.50
invokedyn.	c.s. of $j.$u.invoke.*#*	0	n/a	0	n/a
invokedyn.	invokedynamic c.s.	0	n/a	3	430.67
generics	generic method signature	34	446.26	4	2662.50
generics	generic type signature	32	92.78	4	498.50
generics	generic field signature	33	223.27	4	536.75
generics	generic local variable signature	27	944.85	4	4941.75
dynlang	c.s. of $jx.tools.JavaCompiler#*	0	n/a	1	2.00
dynlang	c.s. of $jx.tools.ToolProvider#*	0	n/a	1	1.00
dynlang	c.s. of $jx.script.*#*	0	n/a	1	37.00
reference	c.s. of $j.$l.ref.WeakReference#*	23	13.04	4	7.50
reference	c.s. of $j.$l.ref.SoftReference#*	11	13.55	2	4.00
reference	c.s. of $j.$l.ref.PhantomReference#*	3	1.00	1	3.00
reference	c.s. of $j.$u.WeakHashMap#*	19	10.42	2	1.00
threads	c.s. of $j.$l.Thread#*	47	12.96	5	4.40
threads	subclass of $j.$l.Thread	39	10.31	1	6.00
threads	impl. of $j.$l.Runnable	44	26.18	5	13.60
threads	c.s. of $j.$u.concurrent.Executors#*	7	1.71	3	3.67
annotation	declares annotation	10	23.30	3	8.33
annotation	uses type annotation	16	78.38	4	235.25
annotation	uses field annotation	10	140.60	4	97.25
annotation	uses method annotation	20	196.10	4	484.25
annotation	uses type use annotation	0	n/a	0	n/a
annotation	uses type parameter annotation	0	n/a	0	n/a
system	native method definition	4	73.25	0	n/a
system	c.s. of $j.$l.Runtime#*	40	13.23	4	6.00
system	c.s. of sun.misc.Unsafe#*	0	n/a	1	30.00
misc	synthetic method definition	68	233.71	6	993.00
misc	bridge method definition	35	115.43	4	1001.50

C Version Policy

1. The first XCorpus release is *version 1.0.0*, this is the version this paper refers to.
2. The micro number is increased when non-functional changes take place, e.g. when new analysis scripts, are added.
3. The minor number is increased when changes are made that affect the coverage of any of the programs, e.g. when new tests or other entry points are integrated, or when tests are re-generated.
4. The major number is increased when new programs (including new program versions) are added to the dataset, or when existing programs are removed from the dataset.

The version info is defined in the version property in *tools/build.xml*. This number is kept consistent with repository tags.

References

[ant] Apache ant. `http://ant.apache.org/`, [accessed 25 May 2016].

[BGH+06] Stephen M Blackburn, Robin Garner, Chris Hoffmann, Asjad M Khang, Kathryn S McKinley, Rotem Bentzur, Amer Diwan, Daniel Feinberg, Daniel Frampton, Samuel Z Guyer, et al. The dacapo benchmarks: Java benchmarking development and analysis. In *Proceedings OOPSLA'06*. ACM, 2006.

[BHM+15] Earl T Barr, Mark Harman, Phil McMinn, Muzammil Shahbaz, and Shin Yoo. The oracle problem in software testing: A survey. *IEEE Transactions on Software Engineering*, 41(5):507–525, 2015.

[Boi16] Ronald F. Boisvert. Incentivizing reproducibility. *Communications of the ACM*, 59(10):5–5, September 2016.

[BOL14] Sushil Bajracharya, Joel Ossher, and Cristina Lopes. Sourcerer: An infrastructure for large-scale collection and analysis of open-source code. *Science of Computer Programming*, 79:241–259, 2014.

[BSSM10] Eric Bodden, Andreas Sewe, Jan Sinschek, and Mira Mezini. Taming reflection (extended version). Technical report, TUD-CS-2010-0066, CASED, 2010.

[CH78] RJ Chevance and T Heidet. Static profile and dynamic behavior of cobol programs. *ACM SIGPLAN Notices*, 13(4):44–57, 1978.

[CP16] Christian Collberg and Todd A Proebsting. Repeatability in computer systems research. *Communications of the ACM*, 59(3):62–69, 2016.

[DER05] Hyunsook Do, Sebastian Elbaum, and Gregg Rothermel. Supporting controlled experimentation with testing techniques: An infrastructure and its potential impact. *Empirical Software Engineering*, 10(4):405–435, 2005.

[DHS15] Jens Dietrich, Nicholas Hollingum, and Bernhard Scholz. Giga-scale exhaustive points-to analysis for java in under a minute. In *Proceedings OOPSLA'15*. ACM, 2015.

[DJB14] Jens Dietrich, Kamil Jezek, and Premek Brada. Broken promises: An empirical study into evolution problems in java programs caused by library upgrades. In *Proceedings CSMR'14*. IEEE, 2014.

[Ern03] Michael D Ernst. Static and dynamic analysis: Synergy and duality. In *Proceedings WODA'03*, 2003.

[FA11] Gordon Fraser and Andrea Arcuri. Evosuite: automatic test suite generation for object-oriented software. In *Proceedings FSE'11*. ACM, 2011.

[FA16] Gordon Fraser and Andrea Arcuri. Evosuite at the sbst 2016 tool competition. In *Proceedings SBST'16*. ACM, 2016.

[git] GitHub. `https://github.com/`, [accessed 9 October 2016].

[GPV01] Christian Grothoff, Jens Palsberg, and Jan Vitek. Encapsulating objects with confined types. In *Proceedings OOPSLA'01*. ACM, 2001.

[Hor97] Susan Horwitz. Precise flow-insensitive may-alias analysis is np-hard. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 19(1):1–6, 1997.

[ivy] Apache ivy. `http://ant.apache.org/ivy/`, [accessed 25 May 2016].

[jac] Jacoco java code coverage library. `http://www.eclemma.org/jacoco/`. Accessed: 2016-09-30.

[JD16] Kamil Jezek and Jens Dietrich. Magic with dynamo–flexible cross-component linking for java with invokedynamic. In *Proceedings ECOOP'16*, volume 56. Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik, 2016.

[jsra] JSR 199: Java Compiler API. `https://jcp.org/en/jsr/detail?id=199` [accessed 12 March 17].

[jsrb] JSR 223: Scripting for the Java Platform. `https://jcp.org/en/jsr/detail?id=223` [accessed 12 March 17].

[kag] Kaggle datasets. `https://www.kaggle.com/datasets`. Accessed: 2016-09-30.

[Knu71] Donald E Knuth. An empirical study of fortran programs. *Software: Practice and Experience*, 1(2):105–133, 1971.

[KV15] Shriram Krishnamurthi and Jan Vitek. The real software crisis: Repeatability as a core value. *Communications of the ACM*, 58(3):34–36, 2015.

[Lic13] M. Lichman. UCI machine learning repository, 2013. URL: `http://archive.ics.uci.edu/ml`.

[LO15] Cristina V. Lopes and Joel Ossher. How scale affects structure in java programs. In *Proceedings OOPSLA'15*. ACM, 2015.

[LSS+15] Benjamin Livshits, Manu Sridharan, Yannis Smaragdakis, Ondrej Lhoták, José Nelson Amaral, Bor-Yuh Evan Chang, Samuel Z Guyer, Uday P Khedker, Anders Møller, and Dimitrios Vardoulakis. In defense of soundiness: a manifesto. *Communications of the ACM*, 58(2):44–46, 2015.

[Mav] Apache maven. `http://maven.apache.org/`, [accessed 25 May 2016].

[MH81] Edward Miller and William E Howden. *Tutorial, software testing & validation techniques.* IEEE Computer Society Press, 1981.

[MRK13] Tyler McDonnell, Baishakhi Ray, and Miryung Kim. An empirical study of api stability and adoption in the android ecosystem. In *Proceedings ICSM'13*. IEEE, 2013.

[Pen11] Roger D Peng. Reproducible research in computational science. *Science*, 334(6060):1226–1227, 2011.

[PM00] Roldan Pozo and Bruce Miller. Scimark 2.0. `http://math.nist.gov/scimark2/credits.html`, 2000. Accessed: 2016-09-30.

[RCV+15] José Miguel Rojas, José Campos, Mattia Vivanti, Gordon Fraser, and Andrea Arcuri. Combining multiple coverage criteria in search-based unit test generation. In *Proceedings SSBSE'15*. Springer, 2015.

[Rep98] Thomas Reps. Program analysis via graph reachability. *Information and software technology*, 40(11):701–726, 1998.

[RvDV16] S Raemaekers, A van Deursen, and J Visser. Semantic versioning and impact of breaking changes in the maven repository. *Journal of Systems and Software*, 2016.

[SCWP09] Kumar Shiv, Kingsum Chow, Yanping Wang, and Dmitry Petrochenko. Specjvm2008 performance characterization. In *Proceedings SPEC Benchmark Workshop*. Springer, 2009.

[sem] Semantic versioning 2.0.0. `http://semver.org/`, [accessed 25 May 2016].

[sou] sourceforge.net. `https://sourceforge.net/`, [accessed 9 October 2016].

[spe] The standard performance evaluation corporation. `https://www.spec.org/`. Accessed: 2016-09-30.

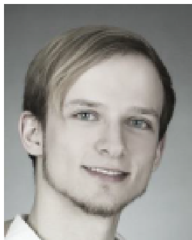
[TAD+10] Ewan Tempero, Craig Anslow, Jens Dietrich, Ted Han, Jing Li, Markus Lumpe, Hayden Melton, and James Noble. The qualitas corpus: A curated collection of java code for empirical studies. In *Proceedings APSEC'10*. IEEE, 2010.

[TMVB13] Ricardo Terra, Luis Fernando Miranda, Marco Tulio Valente, and Roberto S. Bigonha. Qualitas.class Corpus: A compiled version of the Qualitas Corpus. *Software Engineering Notes*, 38(5):1–4, 2013.

About the authors

Jens Dietrich is an Associate Professor at Massey University in New Zealand. Jens' research interests are in the areas of software componentry and evolution and static analysis.

Contact him at j.b.dietrich@massey.ac.nz, or visit https://sites.google.com/site/jensdietrich/.



Henrik Schole studies computer science at the Technical University of Dresden. Henrik created the *ant*-based setup for the XCorpus during an internship at Massey University in 2016.

Contact him at Henrik.Schole@gmx.de.

Li Sui is a PhD student at Massey University. Li's research topic is a study into the unsoundness of static analysis tools.

Contact him at leesui0207@gmail.com.

Ewan Tempero is Associate Professor at the University of Auckland. Ewan's main area of research is of research is measuring software design quality. Ewan manages the original Qualitas Corpus.

Contact him at e.tempero@auckland.ac.nz, or visit https://www.cs.auckland.ac.nz/~ewan/.