Computer Science • 26(4) 2025 https://doi.org/10.7494/csci.2025.26.4.6702

Matija Novak
Marko Mijač

THE BENEFITS OF TESTING SOFTWARE IN SE RESEARCH: LESSONS LEARNED FROM TWO PhD PROJECTS

Abstract *Software engineering (SE) research often involves creating software – either as a primary research output (e.g., in design science research) or as a supporting tool for the traditional research process. Ensuring software quality is essential, as it influences both the research process and the credibility of findings. Integrating software-testing methods into SE research can streamline efforts by addressing the goals of both research and development processes simultaneously. This paper highlights the advantages of incorporating software testing in SE research – particularly for research evaluation. Through qualitative analysis of software artifacts and insights from two PhD projects, we present ten lessons learned. These experiences demonstrate that, when effectively integrated, software testing offers significant benefits for both the research process and its results.*

Keywords software testing, research evaluation, lessons learned, SE research

Citation Computer Science 26(4) 2025: 51–75

1. Introduction

Software engineering (SE) research studies systematic, disciplined, and quantifiable approaches to the development, operation, and maintenance of software [22]. Depending on the overall goal, we can identify two distinct modes of SE research [43]: (1) *solution-seeking* – a pragmatic research that aims to improve the state of affairs by engineering solutions to practical problems; and (2) *knowledge-seeking* – a more traditional research focused on understanding existing phenomena that appear in the software engineering context. Both of these modes of research frequently involve producing software that is of critical importance to research contributions. In solution-seeking research, for example, the produced software may pose as the main research contribution itself (e.g., instantiation artifacts in design science [21]). In knowledge-seeking research, on the other hand, software is often produced to support the research process (so-called *research software*). Whatever the role of the produced software, ensuring appropriate quality is imperative; otherwise, we could compromise the success of research processes and the credibility of research results.

Building working software as a part of a research effort means that, in addition to the usual research activities, the research process now also has to include the requirement specification, design, implementation, and testing of software; i.e., activities that are traditionally part of the software-development process. The result of this is an increase in the amount of work and resource consumption, as our research process has to meet both scientific rigor and practical relevance. In addition, researchers are required to have skills in both traditional research methods and software-development methods to meet these goals. Unfortunately, this is not always the case. Two recent surveys [19, 20] reported that, while 92% of academics use research software and 56% develop their own software, 21% of those have no training in software development. An additional indicator that researchers need to have both research and development skills are recent efforts to form a community of such experts under the term of *research software engineers* [8].

In order to alleviate the complexity and amount of work required when software development is a part of the research process, we could take an approach to direct software-development efforts to simultaneously contribute to research goals. For example, an attempt could be made to find common ground between software testing and the activities of the research process. A particularly promising relationship can be established between software testing and research evaluation, as both activities would be interested in judging the state and the worth of the produced software artifact.

The necessity and benefits of testing in the software-development process are acknowledged by both practitioners and researchers. Testing techniques often drive development efforts (such as in Test-Driven Development (TDD) [5]), they are a prerequisite for the refactoring process [16] and a critical part of DevOps [13]. However, testing is not often employed within the research process and is even less often properly reported. For example, while Hevner et al. [21] listed functional and structural testing (e.g., unit testing) as one of the five design science evaluation methods in their

seminal paper, subsequent literature reviews [1, 12] showed the use of testing to be marginal compared to other evaluation methods.

The goal of this paper is to raise awareness of the benefits that software-testing activities can bring in terms of contributing to both the practical and scientific aspects of software engineering research. We attempt this by qualitatively analyzing the software artifacts and experiences gained from the two software engineering PhD projects that we authored. The results of the analysis are reported in the form of ten lessons learned. These show that, by applying and properly reporting software testing, we can contribute to different aspects of the research process – especially research evaluation. We start the remaining part of the paper with a brief overview of the related literature in Section 2, which is followed by a methodology description in Section 3. In Section 4, we provide detailed information on the analyzed PhD projects and software systems that were produced as a part of them. We report the aforementioned lessons learned in Section 5, and finally offer our conclusions, recommendations, and future research plans in Section 6.

2. Related work

2.1. Software produced in SE research

As suggested in the introductory section of the paper, both modes of SE research can result in software being produced. Regardless of whether the software is the main research result or only has a supporting role, it is often characterized as *scientific software* or *research software*. Kanewala and Bieman [25] described scientific software as software with a large computational component that is usually developed by multidisciplinary teams made up of scientists and software developers. Similarly, Heaton [18] defined scientific software as a "code that is written by scientists for the purpose of doing research." Arvanitou et al. [2] described scientific software development as a process of analyzing, designing, implementing, testing, and deploying software applications for scientific purposes (e.g., physics, biology, medical analysis, and data science).

On the other hand, Hettrick et al. [20] defined research software as software used to generate, process, or analyze results intended to be published in scientific publications. They emphasized that research software can range from a few lines of code written by a researcher himself to a professionally developed software package. Badolato [3] defined research software as being developed to meet specific scientific needs. Such software is developed by scientists as a result of research work and then used by scientists to support their research work. Hettrick et al. [20] further clarified that common software systems that are not created specifically for research purposes should not be called research software. Rather, such software systems can be called software in research [4].

As can be seen, the distinction between scientific software and research software is not strict, and the terms are often used interchangeably. However, it seems that

the term "research software" is broader in scope and is used to refer to any piece of code used to generate, process, or analyze research results. Scientific software is more narrow in scope and usually refers to software performing complex computations in scientific disciplines such as physics, biology, and data science. Throughout the rest of the paper, we will use the term "research software."

2.2. Importance of testing

In 2014 [25], a systematic literature review (SLR) focused on the topic of testing research software was conducted. The authors analyzed 62 studies and found that some studies reported the use of unit testing, system testing, and integration testing. They mentioned that one big problem for testing research software was the oracle problem, meaning that it was not clear what the result should be. However, some studies overcame the problem by generating random test cases with specially designed test cases and similar methods. These studies showed that testing can be done even in extreme cases. They also noted that only two studies reported test coverage and similar information. Even though this is not a problem in itself, it shows that testing is not taken seriously in the process of developing research software.

In addition, this SLR showed that testing was not very common in developing research software and that some reasons for this were objective (such as the oracle problem) while others were not. For example, one reason that was mentioned is that "Scientific software developers are unaware of the need for and the method of applying verification testing" [25] based on [26]. Another reason mentioned is that "Software development is treated as a secondary activity resulting in a lack of recognition for the skills and knowledge required for software development" [25] based on [41]. Such reasons should not be accepted as excuses for not creating tests. Also, such reasons show that the scientific community needs to be informed of the importance of testing and the impacts when testing is neglected.

Another related study was the systematic mapping study by Arvanitou et. al. [2] in 2021. This study showed the use of software engineering practices for scientific application development and the impact on software quality. They analyzed 359 papers and found out that the most of the studied artifacts in SE research were the source codes with 169 papers. On the other hand, only 32 papers dealt with testing. In addition, they stated, "The results suggest that the most commonly reported practices are related to implementation" [2], while testing comes as the fourth-most-common research topic. This showed that there were papers that dealt with testing research software but not as many as were maybe needed. Only seven papers discussed TDD and quality assurance as aspects of testing. On the other hand, a good thing was that "the results showed scientific software-development teams are mostly interested in software implementation and testing activities." [2]

In the PhD thesis from Heaton [18] with the title "Software engineering for enabling scientific software development," the empirical evidence showed that, "by having scientific software-development teams use the software engineering techniques,

this will help researchers evaluate the effectiveness of those techniques for their own use" [18]. However, "while scientific software developers recognize they would benefit from using software engineering practices, those practices that support verification & validation and testing have not been widely adopted" [18].

Another paper that highlighted the importance of testing in the context of performance was [39]. Recently, there has been an increased focus on the use of Large Language Models for testing, as seen in [23, 50].

In order to provide guidelines for the management of scientific digital assets (including research data and research software), Barker et al. [4] introduced the FAIR principles (Findability, Accessibility, Interoperability, and Reusability). These four principles were later expanded by [54], with reviewability and supportability principles. The reviewability principle emphasizes how easily reviewers can examine and evaluate research software. In this process, testing plays a key role, as it helps to demonstrate the correctness of software implementation. Since the exhaustive testing of all inputs and outputs is often impractical, common strategies include equivalence class partitioning, boundary value testing, and leveraging complementary subroutine pairs like "encode-decode" or "serialize-deserialize." Effective testing not only aids in identifying potential errors but also provides reviewers with critical insights into the reliability and correctness of research software.

From the literature, it is evident that testing is represented in scientific research (at least from a theoretical point of view) and that it is considered useful. At the same time, the testing of research software is not widely adopted, and there is very little research focusing on this.

3. Methodology

The authors of this paper submitted and successfully defended their PhD projects. The two PhD projects significantly differed in terms of the topics, research designs, and technologies involved. However, both PhD projects were in the field of software engineering, and they involved the developments of complex software systems.

The first system – a Multiple Plagiarism Checker (MPC) developed in the first PhD project – is an example of research software used to support the research process. On the other hand, the second system was a software framework for managing reactive dependencies in object-oriented (OO) applications (REFRAME), which was developed in the second PhD project; it was a main research contribution in the form of design science instantiation artifact.

The main goal of this research is to describe the experience from these two PhD projects with regard to using software testing in SE research. In accordance with this goal, we pose the following research question:

What are the key lessons learned from using software testing in SE research?

In order to answer this question, the authors performed the following steps:

1. *Analyze:* the authors separately conducted qualitative analyses of their PhD thesis as well as the software artifacts produced during the PhD projects (specifications, designs, source code, tests, etc.).
2. *Extract relevant data:* the authors extracted data relevant to the use of the software testing in their PhD projects.
3. *Identify common themes:* after individual analysis, the authors organized several meetings in which they jointly analyzed and discussed their previous findings and identified themes common for both PhD projects.
4. *Report lessons learned:* the identified themes were compared and supported with experiences from scientific and professional literature and reported in the form of lessons learned.

4. Demonstration cases

In order to clarify the context from which the proposed learned lessons were drawn, we provide details about the analyzed PhD projects and the software systems that were produced as a part of them in this section. Before going into each PhD project in detail, we provide a summary of the demonstration cases in terms of software size, technology used, and test methods used in Tables 1 and 2.

Table 1
Summary of demonstration cases

CASE	Type	Programming language	Test coverage	LOC	Number of classes
MPC	Research software	Java	94%	12.113	240
REFRAME	Software framework	C#	96%	15.000	172
R	Statistical analysis	R	NA	NA	NA

Table 2
Testing methods used

CASE	Testing methods used
MPC	TDD, UnitTests, Integration Test, System Test, Regression Tests, Acceptance Tests
REFRAME	TDD, UnitTests, Integration Tests, System Tests, Regression Tests
R	TDD, UnitTests

4.1. MPC – plagiarism detection

MPC is a research software developed for a PhD project [37] aimed at analyzing the impact of preprocessing techniques on the accuracy of plagiarism detection in

student programming assignments. The experimental design involved running several similarity-detection tools on multiple data sets (each preprocessed with different techniques), followed by statistical analysis using the F1 measure. The study used 6 plagiarism-detection tools, 5 preprocessing techniques, and 133 data sets (44 to 259 files each), resulting in 3990 unique detections. MPC was developed to automate this process, as manually performing these tasks was not feasible.

4.1.1. Description

The first system was developed in Java and featured both CMD and Web GUI interfaces. The Web GUI was built using PrimeFaces on top of the Java Server Faces (JSF) framework. The full system code is available on GitHub (https://github.com/matnovak-foi/MPC).

MPC supports research by preparing and preprocessing data sets, performing detection using external tools, unifying results, calculating statistical measures (e.g., F1, Precision, Recall), and exporting data for further analysis in R.

Execution on a cluster (https://www.srce.unizg.hr/isabella/) with 104 nodes (208 CPUs, 2469 cores, 12.5 TB RAM, 200 TB storage) took around 30 days, generating 1 TB of data. Given its critical role, MPC needed reliability, as failures could compromise months of work. Due to its complexity, automated testing was essential.

4.1.2. Metrics

The code coverage was measured using Clover. With data model classes included, the total coverage was 84.7% (68.1% conditionals, 87.2% statements, and 89.4% methods). Excluding model classes, the coverage increased to 94% (88.3% conditionals, 94.8% statements, and 94.9% methods). GUI methods were not explicitly tested due to their framework-dependent nature but were included in the coverage calculations.

MPC consists of 12,113 lines of code (excluding model classes), with 684 conditionals, 3,679 statements, 1,071 methods, 240 classes, 171 files, and 50 packages. Development, using the Pomodoro technique, took approximately 300 hours (38 workdays), spread over three months.

4.2. REFRAME – software framework

REFRAME is a software framework developed as part of a PhD project using design science research [35]. It aimed to improve the management of reactive dependencies in object-oriented applications. To ensure a rigorous evaluation, the FEDS framework [47] was used to create a detailed strategy aligned with the project goals. The evaluation was conducted in four episodes, starting with formative tests in an artificial setting and concluding with summative tests in a realistic context. The first episode (Prototyping&testing) focused on evaluating REFRAME's efficacy using software testing techniques, which is the relevant episode for this paper.

4.2.1. Description

REFRAME is an object-oriented framework developed in C# that consists of two parts: (1) REFRAME-Core (which handles reactive dependencies), and (2) REFRAME-Tools (a set of tools to enhance usability). The source code is available on GitHub (https://github.com/MarkoMijac/REFRAME).

REFRAME helps manage reactive dependencies by providing abstractions for reactive nodes, which are used to create dependency graphs. These graphs track dependencies between object members and update when changes occur. The REFRAME-Analyzer tool connects to running applications to fetch and analyze dependency graphs, producing a reduced version for visualization in REFRAME-Visualizer as a directed acyclic graph. Over a hundred analyses can be performed and visualized. The framework also optimizes code generation with fluent syntax and code snippets.

4.2.2. Metrics

REFRAME contains 15,000 lines of code across 172 classes and 13 components (.dll files). Approximately 1000 automated tests (20,000+ lines) were written. While the necessary amount of code covered by tests is still an open question, most recommendations in the literature and practice range from 80 to 100%. In order to calculate the code coverage for REFRAME, we used Visual Studio's internal analyzer tool. The calculation is based on block measure, which official documentation for Visual Studio defines as a "piece of code with exactly one entry and one exit point" [34]. The calculated code coverage for the entire framework was 95.7%, while the coverage of the individual components ranged between 91 and 100%. Since all of these numbers were within the upper half of the recommended values, we can conclude that the framework was sufficiently covered with tests.

4.3. Statistical analysis with R

In Section 4.1, an example of research software was presented, and the result was a CSV file for statistical analysis. This can be done using software like Statistica, SPSS, or programming language R (which was used in [37]). R is popular for statistical analysis, offering numerous packages for various analyses. It also allows users to view the source code of functions to ensure accuracy and enables the creation of custom functions for specific analyses.

4.3.1. Description of using R for MPC

In [37], various functions were created for statistical analysis and graphical representations, using data from MPC and a systematic literature review [38]. The experimental design [15,36] involved performing a two-way multifactorial ANOVA to test hypotheses, with contrast comparisons to identify affected groups. Orthogonal contrast and simple effects analysis were also used to further break down interactions.

It is not necessary to explain the statistical analysis in more depth. The point is that a rather extensive statistical analysis was done and not just simple tests like

calculating the mean. When doing such extensive analysis, there are many things that can go wrong because some parameter was set wrong or that the function was called in the wrong way or that there is a logical mistake in a newly created function. So, the problem of ensuring validity is similar here as in any programming any other application.

4.3.2. Metrics

There is no easy way to check the coverage that was done in R; however, there were more than 100 test functions created in the process of doing different kinds of statistical calculations. In the R_UtilityFunctions package, for example, there are 88 tests for 62 functions.

4.4. Testing methods used

The testing for Systems 1 and 2 and the statistical analysis with R followed similar approaches (Fig. 1) to ensure the validity of the software and minimize errors.

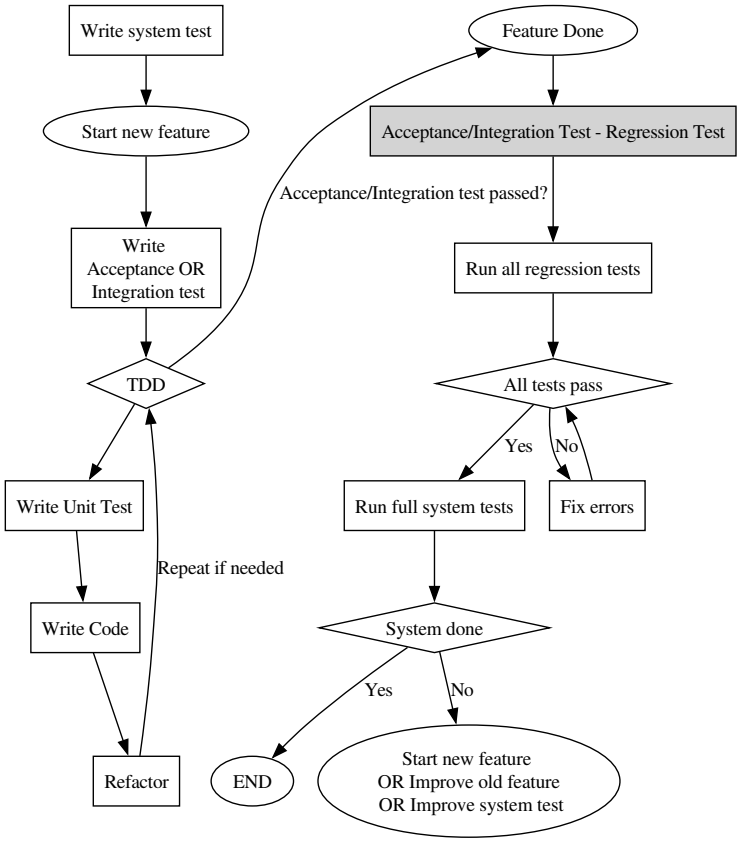

Figure 1. Generalized testing workflow

Test-Driven Development (TDD) [5] was employed in all of the systems, alongside refactoring techniques by Martin Fowler [16], clean code practices by Robert C. Martin [29, 30], and design patterns [17]. Unit tests, the core of TDD, provided immediate feedback during development, following the guidelines of Gerard Meszaros [33]. For MPC, additional acceptance and system tests were written, with acceptance tests later serving as regression tests. Even with a thorough test, a 100% certainty was not guaranteed; however, the tests significantly increased reliability, with no bugs or errors found during execution.

For REFRAME, the testing was formative, guiding the framework's design through TDD. While the tests were written after some exploratory prototyping, the MSTest framework was used to automate the tests, which included unit, integration, and system tests. These results were integrated into corrective and perfective actions to improve the framework.

For the statistical analysis with R, TDD and clean code practices were also applied. Unit tests were written using the *testthat* package http://testthat.r-lib.org/ to check the correctness of the functions that processed quantitative data, such as frequency tables. The graph's appearance was manually checked, and the data input was validated with unit tests. A generic R package, R_UtilityFunctions, was created for future use and is available on GitHub (https://github.com/matnovak-foi/R_UtilityFunctions).

5. Discussion

Software systems produced in analyzed PhD projects are different types of software systems, they are from different domains, and serve different roles within the conducted research. MPC and R statistical analysis from the first PhD project are typical examples of research software. Their role was to support the research process in *"identifying the effect of preprocessing techniques on plagiarism detection accuracy..."*. On the other hand, REFRAME is produced in the second PhD project as a primary contribution; i.e., *a solution for a problem of managing reactive dependencies in applications*. REFRAME is an example of an instantiation artifact, which is one of four main scientific contributions regularly found in design science research.

Now, despite all the differences between the software systems, the analysis that was performed on the written PhD theses produced software artifacts as well as on discussions between the authors; this indicated that there were common themes in both PhD projects related to the role of software testing. In this section, we present these common themes in the form of ten lessons learned.

5.1. LL 1. Testing contributes to evaluation efforts in different types of SE research

As previously described, developing software was a huge part of both PhD projects. The first PhD project conducted an experiment that required a large number of pla-

giarism detections to be performed. Manually performing such a number of analyses was unrealistic, so an *MPC software system* was developed to support the experiment process and automate the aforementioned analyses.

The second PhD project conducted design science research in which a software framework (*REFRAME*) was developed as a form of instantiation artifact intended to solve a primary problem stated in the dissertation. Both PhD projects heavily employed automated testing during the software-development process. In the authors' experience, this resulted in multiple benefits for both the overall research process and the research outcomes.

The two PhD projects can be seen as representatives of two distinct types of SE research: *knowledge-seeking*, and *solution-seeking* research [43].

Knowledge-seeking research is a traditional research primarily aimed at investigating existing phenomena that appear in a software engineering context (e.g., investigating what effect preprocessing techniques have on plagiarism-detection accuracy). In this type of research, the developed software can be characterized as research software and has a supportive role; i.e., it supports the activities of the research process (e.g., gathering data, extracting data, transforming data, analyzing data, etc.). Proper testing increases the chance that the software is correct; i.e., that it correctly performs its supporting function. In this way, the testing indirectly participates in the research evaluation, as it makes the research process more reliable and the research results more credible. For example, if the software is intended to collect data that will serve as a basis for developing models and answering research questions, then software testing increases the chance that we obtained the right data.

Solution-seeking type of research, on the other hand, is more pragmatic than knowledge-seeking research. It intervenes in the software engineering context and tries to improve the state of affairs by engineering solutions to practical problems (e.g., by creating specific software frameworks intended to facilitate the management of reactive dependencies in OO applications). In this type of research, the developed software is the very solution we engineered to tackle the stated problem, which makes it a primary research output (rather than a supportive one). That being the case, software testing now directly participates in research evaluations. First, during the development, the testing provides early feedback on the state and correctness of the software. This gives developers a chance to resolve issues and improve software while this is still feasible (the so-called formative evaluation). After the development is done, we use testing to ensure that the software indeed does what it is required to do and that it does it correctly, thus evaluating the worth of the software (the so-called summative evaluation).

Regardless of the type of SE research and the role of the developed software, the underlying research process and research outcomes greatly depend on the quality of the developed software. By incorporating software tests in the research process, we improve the software quality and provide empirical evidence for it – thus benefiting evaluation efforts.

5.2. LL 2. TDD provides guidance for software-design activity

MPC and REFRAME as software solutions played a critical role in the practical and scientific contributions of the two PhD projects. Therefore, it was essential from the very start to design these solutions in a quality manner. In both of our experiences, the application of TDD had a significant role in achieving that. Writing tests before code helped the authors to first adopt an external code consumer's view of a particular code instead of prematurely jumping right into technical and implementation details, with the additional emphasis on code testability and building a system that is testable by design. With TDD, the authors were forced to think in terms of smaller but more cohesive functionalities, thus better separating concerns and improving overall code quality. Having such cohesive code elements, it was easier for the authors to assemble their respective software solutions in the first place, but also to maintain the software afterwards. One big benefit to the authors was that there was no fear of changing the system and improving the design when it was necessary.

There was an additional benefit of the test-first approach in the second PhD project. REFRAME is a software framework whose natural "user interface" is a set of available types and their exposed members; i.e., API. By applying a test-first approach, the author was also able to first imagine the framework's API through the eyes of potential users (application developers). This provided additional inputs in terms of framework usability early on in the development process and led to the further optimization and fine-tuning of the framework's API. This was extremely important, as the steep learning curve is one of the most recognized barriers in using frameworks.

Software design plays a decisive role in software quality. Whether the software is a direct result of SE research or a tool developed to support the research, the quality of the software will affect the success of the research itself. Quite a lot of research indicates a positive impact TDD has on software quality. For example, in their systematic review, Bissi et al. [6] reported that 76% of the identified studies demonstrated a significant increase in internal software quality and 88% external software quality as a consequence of the application of TDD. Our experiences are in line with these reports, as TDD provided valuable guidance in our efforts to design MPC and REFRAME.

5.3. LL 3. Testing is safety mechanism when making changes in software

When developing software, it is not possible to know everything at the beginning. Even with the best planning, there is always something that changes during the course of development. There is always the factor of the unknown. This is especially true for research software because in science we deal with the "unknown" to find new knowledge. Therefore, research software is prone to change at the last moment when we discover some new facts. Changing software at the last minute is risky, but with a test suite in place, it is not impossible. Tests that were written during development now become regression tests and give a safety net and confidence to the developer

that, by doing the necessary changes, one will not break the code in some unexpected places. This is confirmed in the three demonstration cases.

One such event during the implementation of MPC was when the idea happened to pass data to a phase. To explain what happened, one first needs to understand in a bit more detail how the system was built. MPC has three main execution phases. The first phase is the preparation of data, which includes the following preparations: extracting the data set from the archive; renaming all folders to unique names; and delete all files that are not used in the detection process like images, pdf files, etc. The next phase is the preprocessing phase, where the different preprocessing techniques are applied on each individual data set. The third and final phase is the detection phase, where similarity detection was performed in each preprocessed data set using the six different tools. Now, first, each component was developed separately, which means that first all preparation methods, preprocessing techniques, and adapters for the tools were developed. Then, the different components were integrated into a single process using a class called PhaseRunner. PhaseRunner, at first, had the job only to start the phases in correct order and nothing more. But soon it was clear that there needs to be one folder for each data set where the different executions need to be performed and that the PhaseRunner is also responsible for telling the individual phase what the working directory is. This means each phase will return the output directory where the results are stored and accept an input directory which to use when the phase is run.

Here is where the problems started, because there was no way to pass in such information; this meant it needed to be added to all components. This seems simple enough that it can be done without problems, but it was not. In fact, adding this option was more difficult than expected and required changing around 40 classes. The reason why so many classes needed to be changed was that this was a fundamental change to the logic of how the system would operate. Thankfully, there were tests that showed where the places of error were and informed about the problems right away. Otherwise, these problems would be caught much later – maybe only at the end – and then fixing them would be much more difficult. Probably, without the test, this improvement would never be made and would lead to much worse system design.

The tests also played a pronounced role as a safety mechanism in the second PhD project. The development of REFRAME as a novel framework was characterized by a high level of uncertainty. Until the very end, it was not evident what design, implementation, and underlying technologies would make the final solution. Because of this, frequent prototyping sessions were conducted in order to try out different design, implementation, and technology options. Having a large test suite made it possible to freely experiment without the fear of introducing subtle errors in the framework.

Regression tests are the key to enable change and the maintenance of source-code. This was already established in software engineering more than 10 years ago [49]. According to [14], there are 28 empirically evaluated techniques on regression testing.

Unfortunately, according to the mapping study done by Arvanitou et. al. [2] in 2021, only 5% (19 of 359) of the studies investigated the practice of testing, including regression testing. The number was slightly better in [14], where 27 papers were identified. In comparison to the number of papers in the area of programming, this was not enough.

5.4. LL 4. Testing helps in handling complexity

One of the early setbacks that the authors faced during the development of MPC and REFRAME was due to complexity of their intended solutions. Being eager to jump into creating their software solutions, both authors initially started writing code without tests. However, during the course of a few weeks, it became increasingly difficult to continue evolving the solution, as we managed to turn an already complex problem domain into even more complex programming code. Due to so many things that had to be done, bugs that manifested at random times and places, and subtle details that required our attention, we became overwhelmed with complexity thrown at us and got stuck in the development process. In addition, there was constant fear of how to be sure that there will not be problems in the final phase of the research. Since there were no tests, refactoring was difficult and neglected, so the code became more and more unclean and difficult to maintain.

To mitigate that, we started over – this time by applying TDD. Now, the unit tests became foundational blocks of our software solutions. Being by definition a type of test that evaluated individual and isolated units of software, the unit tests served as a mental tool for decomposing complex problems into smaller problems. Now, we were able to shift our focus on one manageable thing at a time and progress with our solutions in small but certain increments. This gave us more confidence; with time, the progress was faster instead of slower as in the first attempt. The refactoring was done all the time and without fear; therefore, the code was clean and easy to maintain.

Developing software is a difficult endeavor. Many influential authors such as Grady Booch [7] seek causes for this in the increasingly complex problem domain we are dealing with as well as inherent complexity of software as a solution. When software development is carried out as part of research, it becomes much more complicated because of the additional requirements that scientific rigor may place on the software-development process. Even a full-time software engineer may find this too much to handle, let alone an SE researcher with many other duties. Unit testing and TDD can help mitigate this problem by serving as a natural decomposition tool.

5.5. LL 5. Testing evaluates scientifically relevant properties

Although the unit testing framework was a key piece of testing technology used by both authors, not all tests created using this framework were unit tests. Instead, we used a technique known as the testing pyramid. The testing pyramid's base was made up of unit tests, which were the most prevalent tests concentrated on individual

units of the code. After that, a smaller number of integration tests were written to determine whether the separate units worked properly when combined. Last but not least, a small number of system tests were created to examine the key features of the software solutions as a whole.

In today's software development, writing tests on these three levels before we hand it over to users is crucial. However, these tests can also help to evaluate the research of certain software properties. For example, Prat et al. [40] studied design science research articles to assemble a list of frequently evaluated artifact properties. They discovered that **efficacy**, or "the degree to which the artifact fulfills its goals considered narrowly without addressing situational concerns" was by far the most often evaluated property. This was precisely the case in the second PhD project. As a formal part of the REFRAME evaluation, testing was used to evaluate early efficacy under laboratory conditions. The tests provided evidence that the framework and its parts work as expected and can be used to handle reactive dependencies. The testing took a similar role also in the first PhD project; however, it was not a formal part of the research evaluation.

In addition to efficacy (although not demonstrated in our PhD projects), testing could be a suitable method to evaluate some other properties from the list that Prat et al. [40] assembled, including accuracy, validity and completeness.

5.6. LL 6. Testing should be combined with other evaluation methods to provide comprehensive evaluation strategy

Although we advocate for the use of testing as an evaluation method in SE research, testing should not be the sole evaluation method used in a research. Rather, testing should be an integral part of the comprehensive evaluation strategy (Fig. 2) performing multiple evaluation methods. For example, in the second PhD project, testing was used in the first evaluation episode to formatively evaluate efficacy in artificial (laboratory) conditions. This was supplemented by three summative evaluation episodes that gradually moved REFRAME from the laboratory to the real world.

When using testing as part of an evaluation strategy, we have to be aware of both its strengths and limitations and supplement it with other evaluation episodes that overcome these limitations. One of the tools that can help us design evaluation strategies but also discuss the role of testing within these strategies is the framework for designing evaluation strategies proposed by Venable et al. [47]. They depicted the evaluation strategy as a trajectory of evaluation episodes placed within two dimensions: (1) functional purpose (formative vs summative evaluation), and evaluation paradigm (artificial vs naturalistic). Formative evaluation is done before an artifact is complete in order to provide feedback that guides further development. Summative, on the other hand, is conducted after the artifact is complete, with the purpose of assessing its overall value. With respect to the evaluation paradigm, artificial evaluation is done in controlled laboratory conditions, while naturalistic evaluation assumes evaluation in a realistic setting and conditions and done by real users.

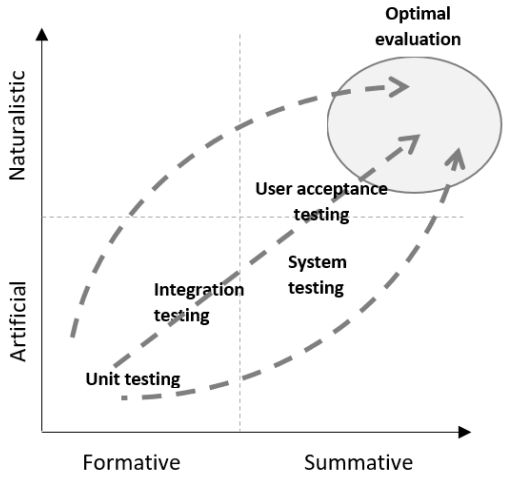

Figure 2. Role of testing in evaluation strategy

The ultimate goal of research software evaluation within SE research would be to provide summative naturalistic evaluation; i.e., thoroughly evaluate software in its entirety in perfectly realistic conditions. However, this can be costly, risky, and sometimes even not possible. This is why evaluation is often done in steps, with evaluation trajectory starting in an artificial setting with a formative role and gradually moving toward a more summative role and more naturalistic conditions. Evidently, by its definition, testing fits into the artificial-evaluation paradigm, as its very purpose is to evaluate software before it is put to real use. Some testing methods such as unit testing and integration testing take formative roles, as they are done in parallel with development. Others such as system testing, E2E testing, acceptance testing, and non-functional testing can also take the role of summative evaluation, as they require software to be complete to give realistic assessment.

However, testing alone cannot capture all aspects of software validation [11, 42] – particularly those influenced by real-world factors. For example, empirical studies such as case studies or surveys can provide broader insights into how software is actually used in practice, offering a more holistic understanding of its value in different contexts. Prototyping and simulation serve as valuable supplements to testing, allowing developers to explore new ideas in a controlled manner before committing to full-scale implementation. While testing may confirm that a component functions as intended, prototyping helps to identify potential problems early by experimenting with different configurations or system behaviors. In addition, peer reviews and expert evaluations focus on the qualitative aspects of the system such as code quality, maintainability, and architectural decisions. These methods can identify potential issues that testing alone may overlook, such as inefficient algorithms or design flaws that might impact scalability or ease of future development.

Although it is hard to prescribe exact evaluation episodes that are to be combined with testing episodes, researchers should strive to fill the gap that testing methods leave in terms of naturalistic evaluation. This means that the testing episode should be accompanied by at least one (non-testing) summative naturalistic evaluation episode. Depending on software maturity, this evaluation episode could target one, two, or even all of the three reality dimensions; i.e., real users, real problem, and real setting. For example, one can apply software to a real problem in laboratory conditions, give software to real users to solve illustrative examples, or jointly conduct action research with real practitioners in a real company. In addition, it is always a good option to use a mixed-method approach to strengthen evaluation. So, for example, if we in one evaluation episode provided tests and coverage analytics to quantitatively demonstrate our software's correctness, another episode could use a focus group to investigate the perception of our users.

When designing comprehensive evaluation strategies in SE, we can choose from a wide range of evaluation methods to complement testing. In case of high-assurance systems where correctness is critical, testing as an evaluation method might not be reliable enough. In such cases, we might opt for more-formal methods such as formal verification [44] to provide mathematical proof of software correctness. In performance-critical domains, benchmarking [24] can complement testing by offering performance comparisons, helping to position the software within the context of competing systems or industry standards. Additional proof that developed software achieves its desired effects can be provided by conducting experiments [53]. In order to evaluate software in real contexts and from the perspective of real users, qualitative methods such as case studies [27], action research [52], and focus groups [46] can be used.

5.7. LL 7. Tests provide proof (evidence) of software correctness

In science, the results must be precise and correct so that valid conclusions can be drawn. In most research, some kind of software is used, and the researcher has to trust that software works as it is supposed to. When external tools are used by many researchers or users (e.g., SPSS, SAS), one can trust that the software works correctly by looking at bug reports and experiences from other users. But when a software is used only by a few people or one person, how do we even know that it works correctly?

When using or developing software that is a part of research, it should therefore be mandatory to provide proof that the software works correctly. If it is not mandatory, a totally wrong conclusion can be made because of an error in the software that was used.

In practice (professional development [5,32]), testing software is a discipline that should provide such proof. Based on the literature [2,18,25], it is evident that testing has a positive impact on developing software used in scientific research and that should be encouraged; these articles also show that it is not used all the time. That testing has a positive impact can be seen when analyzing the three cases, and it confirms that tests provide a proof of correctness.

But what kind of testing should researchers do when developing their software or require from developers if someone else is developing it? Also, how should this proof be reported in the paper? The reporting will be discussed in a separate lesson, so let us focus on the first question.

Based on the three cases, we propose the following. First, we propose to use unit testing since it is a discipline used in practice [32] and has been proven to be effective [45]. Unfortunately, unit tests are very low-level, and one has to have a good idea of what the software should do; so, it is recommended to start by writing an Acceptance Test or Full system test (also called end-to-end tests). This test basically describes the main idea of what a tool should do. Naturally, this might change over the course of time, but it is still a way of expressing what the purpose of the system is.

These acceptance and full-system tests help in cases where unit testing might not be possible due to the nature of the research. Nonetheless, we believe that unit tests should be used for developing research software or software parts that have concrete input and output data. Even in cases where the data is vague, one can write tests that test the results approximately. In such cases, tests are not exact regarding the final usage but prove that the component connections of the system are working as intended.

5.8. LL 8. Report quantitative metrics (e.g., code coverage) related to testing

It might happen that testing was used while developing research software; however, no report is given (since the author does not believe that it is required or necessary). In this paper, we tried to provide concrete use cases that showed the importance of testing as well as examples of how testing was reported when doing research.

Reporting test-code coverage is a good first step. It is known [31] that code coverage is not perfect, but it gives first insight into the amount of testing that was done. There is a study [28] that showed that code coverage is moderately or strongly correlated to the effectiveness of a test suite. In the case of both MPC and REFRAME, the authors have provided not only an overall code coverage but code coverage based on each module; this provides a better overview of the parts that were or were not tested. Now of course this data is always questionable; therefore, Lesson 9 about the importance of source code availability and open data comes into play.

5.9. LL 9. Source code, tests, and test data should be made available if possible

This LL might be obvious, but sometimes researchers do not publish their programs and data. This is problematic since other researchers cannot replicate the results. It would be best if it was possible to provide code and data. By just providing the source-code, one can already use the system on different data and by providing the data so others can test it with a different tool (if one exists).

Providing data that was used in research can be viewed in two ways. One is data used for the tests themselves. When someone has a test that uses data that is not provided, such a test will fail; with tests, it should be mandatory to provide the data that it uses. The data for the tests include test scenarios, test plans, test documentation, and any other documentation that the author used.

On the other hand, by providing data used for the research itself (and not just test data), it becomes possible for others to verify that there is no bias in the data for that tool, that there are no errors in the data itself, or check if this tool is really better than others (in the cases when research performs comparisons).

All of this source-code, test data, and research data enables us to control factors that impact the research results and provide more-objective comparisons between different tools, approaches, etc. In the literature, there are different studies [9, 48] that have dealt with the importance of open access, open source, and other areas that confirm this lesson.

Finally, this lesson is also supported by FAIR principles, which were introduced by Barker et al. [4]. By complying with FAIR, software artifacts produced as a part of a research process are easier to find, and they are more accessible, interoperable, and reusable. This significantly increases the transparency and reproducibility of the research.

5.10. LL 10. Testing is beneficial – even when performing statistical analysis

In the third case (about R), testing was used while writing our own statistical functions. In this case, one can see that testing (in this case, unit testing) goes beyond the software engineering field. Many researchers in different fields use R and write their own R functions. The same problems that we discussed in the other lessons apply here. R functions are specific for statistical analysis; but in their essence, they are just software programs, so they should not be treated differently.

If a function is used by many users, we have some assurance that it is doing the right thing; if not, then the same problems occur that if there is an error. R functions are great in the sense that they are always open source, so the code can be examined. By having tests as well, one can quickly check if the function works as the author intended, and one can check if there was a flaw in the reasoning of the author when building these functions.

There were some personal situations where using tests was shown to be very useful, and not having tests was a bad idea. The first example was when a graph was created to display the number of articles that dealt with plagiarism-detection tools per year. At first, it seemed like a very simple graph, and it was a relatively simple function to display it. Now, since it was a graph, no tests were used. By doing some other graphs, it was discovered that some data was wrong. Since it was not clear where the error was, a test was created for this function (or to be more precise, a part that creates the data that needs to be displayed). By doing the test, the error was

quickly fixed since the problem was that NA values were taken into account rather than being dismissed. Now, if the test was done in the first place, the error would have never occurred; it was only by pure luck that it was discovered that something is wrong with the graph. It might not seem reasonable how such errors can be missed, but when there is a lot of data, it is very difficult to know if some number is off by little; by having tests, it will ensure that such mistakes are recognized quickly.

Another example was when additional logic was added to the function that manipulates data frames (a data structure used by R). At first, there was the need to create a frequency table; but then, it was necessary to have the creation of a frequency table for a list of variables. It is not relevant what exactly the function does. In this case, tests were done; when the extension was implemented at some point, the creation of a frequency table without a list of variables stopped working. This was a quick fix; but if the test had not been there, it would have been a big problem. The reason is that some data was returned that was wrong. Hopefully, this shows how useful unit tests are in writing statistical methods.

One has to admit that writing tests for statistical calculations is a bit different from normal programming, but it has the same effect. In fact, it is easier to write tests for statistical methods since one can create test input data for which it is known what the output of the function should be. Now, easy does not imply that every statistical function is simple but rather that the process is very defined and concrete, which enables writing concrete tests. The difficult part in writing tests for statistical functions is that, sometimes, there is an unlimited number of data combinations, so there is no way to test every possible combination. By testing the degenerate cases and some common cases of usage, one can get quite confident that the function behaves as it should.

In [51], it was explained why writing tests was important. The benefits included decreased frustration, better code structures, fewer struggles to continue development after a break, and increased confidence. For more information on testing R code, there is a book [10] with the exact title.

6. Conclusion

In this study, we have demonstrated that any kind of software engineering research can benefit from applying testing (with a focus on unit tests). Although there are areas where it might be more difficult to write tests, one should try to overcome this problem and find a way.

By writing unit tests or tests in general, one is forced to think about the outcome and what the software should do rather than focusing on how to do it. Therefore, it helps to think about the problem. By writing tests, one thinks about the software twice; i.e., one time about the desired outcome, and the other about how to implement what is desired. Also, anytime the program needs to change, the tests will make sure that nothing was accidentally broken in the process.

By having tests, one then provides at least some proof that others can trust that the results of the system are correct and that the system works as expected. Without tests, there is a higher probability that the system has some part(s) that do not work as expected. It is always the question of how far to go in the testing; for this, (unfortunately) there is no simple answer. All it can be said is that it is better to have a test than not. Excuses for not using tests because the lack of time or knowledge is wrong and should not be excused. From the experience in these projects, one can say that tests only make you go faster in the end.

There are still open questions like "What metrics should be reported?" or "What level of code-coverage is sufficient?" Answering these questions is beyond the scope of this research and needs to be addressed in future work. We can only say that some kind of quantitative metric should be reported anytime a piece of software is developed as part of research. Also, regarding code-coverage, there is a simple answer: only 100% code coverage is enough; however, this is not feasible to achieve. Based on the experience of the authors, we might say that above 90% is usually a good-enough number if the 10% is not the core logic of the program.

While reporting is important in order to be able to evaluate what is stated, one would need to open-source the code, tests, and data to have a complete review and reproducible research. In addition, research software must conform to the FAIR principles in order to be more transparent, reproducible, and reusable.

In addition, it was shown that testing goes beyond the field of software engineering and can be useful where one has to write source code. Since R is a statistical programming language and statistics is used in every field, it is safe to say that testing is useful in all research fields.

Software testing is not the only way of validation and it does not cover all aspects of software validity – especially those that are influenced by real-world factors. Combining testing with other methods such as formal verification, researchers can ensure that a system not only works as expected but also conforms to its formal specifications. Nevertheless, based on the benefits that testing provides, it is suggested that testing should be mandatory in every software-development process and, therefore, in all scientific research where new software is developed.

References

[1] Arnott D., Pervan G.: Design science in decision support systems research: An assessment using the Hevner, March, Park, and Ram Guidelines, *Journal of the Association for Information Systems*, vol. 13(11), p. 1, 2012. doi: 10.17705/1jais.00315.

[2] Arvanitou E.M., Ampatzoglou A., Chatzigeorgiou A., Carver J.C.: Software engineering practices for scientific software development: A systematic mapping study, *Journal of Systems and Software*, vol. 172, 110848, 2021. doi: 10.1016/j.jss.2020.110848.

[3] Badolato A.M.: Research software as a pillar of open science, 2022. https://www.ouvrirlascience.fr/research-software-as-a-pillar-of-open-science.

[4] Barker M., Chue Hong N.P., Katz D.S., Lamprecht A.L., Martinez-Ortiz C., Psomopoulos F., Harrow J., Castro L.J., Gruenpeter M., Martinez P.A., Honeyman T.: Introducing the FAIR Principles for research software, *Scientific Data*, vol. 9(1), p. 622, 2022. doi: 10.1038/s41597-022-01710-x. Publisher: Nature Publishing Group.

[5] Beck K.: *Test-driven development: by example*, Addison-Wesley Professional, 2003.

[6] Bissi W., Serra Seca Neto A.G., Emer M.C.F.P.: The effects of test driven development on internal quality, external quality and productivity: A systematic review, *Information and Software Technology*, vol. 74, pp. 45–54, 2016. doi: 10.1016/j.infsof.2016.02.004.

[7] Booch G., Maksimchuk R.A., Engle M.W., Young B.J., Conallen J., Houston K.A.: *Object-Oriented Analysis and Design with Applications*, Addison-Wesley Professional, Upper Saddle River, NJ, 3rd ed., 2007.

[8] Brett A., Croucher M., Haines R., Hettrick S., Hetherington J., Stillwell M., Wyatt C.: Research software engineers: state of the nation report 2017, 2017.

[9] Corrado E.M.: The Importance of Open Access, Open Source, and Open Standards for Libraries: Theme: Open Access Journals, *Issues in Science and Technology Librarianship*, vol. Spring 2005(42), 2005. doi: 10.29173/istl2002.

[10] Cotton R.: *Testing R Code*, Chapman and Hall/CRC, 2017. doi: 10.1201/9781315380285.

[11] Dasso A.: *Verification, validation and testing in software engineering*, IGI global, 2006.

[12] Deng Q., Wang Y., Ji S.: Design science research in information systems: A systematic literature review 2001-2015. In: *International Conference on Information Resources Management (CONF-IRM)*, 2017.

[13] Ebert C., Gallardo G., Hernantes J., Serrano N.: DevOps, *Ieee Software*, vol. 33(3), pp. 94–100, 2016. doi: 10.1109/ms.2016.68.

[14] Engstrom E., Runeson P., Skoglund M.: A systematic review on regression test selection techniques, *Information and Software Technology*, vol. 52(1), pp. 14–30, 2010. doi: https://doi.org/10.1016/j.infsof.2009.07.001.

[15] Field A., Miles J., Field Z.: *Discovering statistics using R*, SAGE Publications Ltd, 2012.

[16] Fowler M.: *Refactoring: improving the design of existing code*, Addison-Wesley Professional, 2018.

[17] Gamma E., Helm R., Johnson R., Vlissides J.: *Design patterns: elements of reusable object-oriented languages and systems*, Addison-Wesley Reading, 1994.

[18] Heaton D.: *Software engineering for enabling scientific software development*, Ph.D. thesis, University of Alabama Libraries, 2015.

[19] Hettrick S.: softwaresaved/software_in_research_survey_2014: Software in research survey, 2018. doi: 10.5281/zenodo.1183562.

[20] Hettrick S., Antonioletti M., Carr L., Chue Hong N., Crouch S., De Roure D., Emsley I., *et al.*: UK Research Software Survey 2014 [dataset], 2014. doi: 10.5281/zenodo.14809.

[21] Hevner A.R., March S.T., Park J., Ram S.: Design science in information systems research, *MIS quarterly*, pp. 75–105, 2004. doi: 10.2307/25148625.

[22] IEEE Standard Glossary of Software Engineering Terminology, *IEEE Std 61012-1990*, pp. 1–84, 1990. doi: 10.1109/IEEE.1990.101064. Conference Name: IEEE Std 610.12-1990.

[23] Jalil S., Rafi S., LaToza T.D., Moran K., Lam W.: ChatGPT and Software Testing Education: Promises & Perils. In: *2023 IEEE International Conference on Software Testing, Verification and Validation Workshops (ICSTW)*, pp. 4130–4137, IEEE, 2023. doi: 10.1109/icstw58534.2023.00078.

[24] John L.K., Eeckhout L.: *Performance evaluation and benchmarking*, CRC Press, 2018.

[25] Kanewala U., Bieman J.M.: Testing scientific software: A systematic literature review, *Information and Software Technology*, vol. 56(10), pp. 1219–1232, 2014. doi: 10.1016/j.infsof.2014.05.006.

[26] Kelly D., Sanders R.: The challenge of testing scientific software. In: *Proceedings of the 3rd annual conference of the Association for Software Testing (CAST 2008: Beyond the Boundaries)*, pp. 30–36, 2008.

[27] Kitchenham B., Pickard L., Pfleeger S.L.: Case studies for method and tool evaluation, *IEEE Software*, vol. 12(4), pp. 52–62, 1995. doi: 10.1109/52.391832.

[28] Kochhar P.S., Thung F., Lo D.: Code coverage and test suite effectiveness: Empirical study with real bugs in large systems. In: *2015 IEEE 22nd International Conference on Software Analysis, Evolution, and Reengineering (SANER)*, pp. 560–564, IEEE, 2015. doi: 10.1109/saner.2015.7081877.

[29] Martin R.C.: *Clean code: a handbook of agile software craftsmanship*, Pearson Education, 2009.

[30] Martin R.C.: *Clean Architecture: A Craftsman's Guide to Software Structure and Design*, Robert C. Martin Series, Pearson Education, 2017.

[31] Martin R.C.: *Clean agile: back to basics*, Pearson Boston, 2019.

[32] Martin R.C.: *Clean Craftsmanship: Disciplines, Standards, and Ethics*, Pearson Education, Limited, 2021.

[33] Meszaros G.: *xUnit test patterns: Refactoring test code*, Pearson Education, 2007.

[34] Microsoft: Code coverage testing - Visual Studio, 2024. https://docs.microsoft.com/en-us/visualstudio/test/using-code-coverage-to-determine-how-much-code-is-being-tested.

[35] Mijač M.: *Design and evaluation of software framework that improves the management of reactive dependencies in development of object-oriented applications*, Ph.D. thesis, University of Zagreb Faculty of Organization and Informatics, 2021. https://urn.nsk.hr/urn:nbn:hr:211:897057.

[36] Milliken G., Johnson D.: *Analysis of messy data Volume 1 Designed Experiments*, CRC Press, 2009.

[37] Novak M.: *Effect of source-code preprocessing techniques on plagiarism detection accuracy in student programming assignments*, Ph.D. thesis, University of Zagreb Faculty of Organization and Informatics, 2020. https://urn.nsk.hr/urn:nbn:hr:211:528052.

[38] Novak M., Joy M., Kermek D.: Source-code Similarity Detection and Detection Tools Used in Academia: A Systematic Review, *ACM TOCE*, vol. 19(3), 2019. doi: 10.1145/3313290.

[39] Pargaonkar S.: A comprehensive review of performance testing methodologies and best practices: software quality engineering, *International Journal of Science and Research (IJSR)*, vol. 12(8), pp. 2008–2014, 2023. doi: 10.21275/sr23822111402.

[40] Prat N., Comyn-Wattiau I., Akoka J.: A Taxonomy of Evaluation Methods for Information Systems Artifacts, *Journal of Management Information Systems*, vol. 32(3), pp. 229–267, 2015. doi: 10.1080/07421222.2015.1099390.

[41] Segal J.: Some problems of professional end user developers. In: *IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC 2007)*, pp. 111–118, IEEE, 2007. doi: 10.1109/vlhcc.2007.17.

[42] Srinivasan K., Devi T.: Software metrics validation methodologies in software engineering, *International Journal of Software Engineering & Applications*, vol. 5(6), p. 87, 2014. doi: 10.5121/ijsea.2014.5606.

[43] Stol K.J., Fitzgerald B.: Guidelines for Conducting Software Engineering Research. In: *Contemporary Empirical Methods in Software Engineering*, pp. 27–62, Springer, 2020. doi: 10.1007/978-3-030-32489-6_2.

[44] Todorov V., Boulanger F., Taha S.: Formal verification of automotive embedded software. In: *Proceedings of the 6th Conference on Formal Methods in Software Engineering*, pp. 84–87, 2018. doi: 10.1145/3193992.3194003.

[45] Tosun A., Ahmed M., Turhan B., Juristo N.: On the effectiveness of unit tests in test-driven development. In: *Proceedings of the 2018 International Conference on Software and System Process*, pp. 113–122, 2018. doi: 10.1145/3202710.3203153.

[46] Tremblay M.C., Hevner A.R., Berndt D.J.: Focus groups for artifact refinement and evaluation in design research, *Communications of the association for information systems*, vol. 26(1), p. 27, 2010. doi: 10.17705/1cais.02627.

[47] Venable J., Pries-Heje J., Baskerville R.: FEDS: a Framework for Evaluation in Design Science Research, *European Journal of Information Systems*, 2014. doi: 10.1057/ejis.2014.36.

[48] Von Krogh G., Spaeth S.: The open source software phenomenon: Characteristics that promote research, *The Journal of Strategic Information Systems*, vol. 16(3), pp. 236–253, 2007. doi: 10.1016/j.jsis.2007.06.001.

[49] Wahl N.J.: An Overview of Regression Testing, *ACM SIGSOFT Software Engineering Notes*, vol. 24(1), pp. 69–73, 1999. doi: 10.1145/308769.308790.

[50] Wang J., Huang Y., Chen C., Liu Z., Wang S., Wang Q.: Software testing with large language models: Survey, landscape, and vision, *IEEE Transactions on Software Engineering*, 2024. doi: 10.1109/tse.2024.3368208.

[51] Wickham H.: testthat: Get started with testing, *R Journal*, vol. 3(1), 2011. doi: 10.32614/rj-2011-002.

[52] Wieringa R., Moralı A.: Technical action research as a validation method in information systems design science. In: *International Conference on Design Science Research in Information Systems*, pp. 220–238, Springer, 2012. doi: 10.1007/978-3-642-29863-9_17.

[53] Wohlin C., Runeson P., Höst M., Ohlsson M.C., Regnell B., Wesslén A., *et al.*: *Experimentation in software engineering*, vol. 236, Springer, 2012. doi: 10.1007/978-3-662-69306-3.

[54] Zhang H., Maillo A., Khan S.A., Martínez-de Morentin X., Lehmann R., Gomez-Cabrero D., Tegnär J.: Reviewability and supportability: New complementary principles to empower research software practices, *Computational and Structural Biotechnology Journal*, vol. 23, pp. 3989–3998, 2024. doi: 10.1016/j.csbj.2024.10.034.

Affiliations

Matija Novak
University of Zagreb, Faculty of Organization and Informatics, Pavlinska 2 42000 Varaždin, Croatia, matija.novak@foi.unizg.hr

Marko Mijač
University of Zagreb, Faculty of Organization and Informatics, Pavlinska 2 42000 Varaždin, Croatia, marko.mijac@foi.unizg.hr

Received: 06.11.2024
Revised: 29.01.2025
Accepted: 08.04.2025