Is cleaner greener? On the energy impact of refactored test smells

Master's thesis

Simon Biennier

Is cleaner greener? On the energy impact of refactored test smells

THESIS

submitted in partial fulfillment of the
requirements for the degree of

MASTER OF SCIENCE

in

COMPUTER SCIENCE

by

Simon Biennier
born in Ambilly, France

Software Engineering Research Group
Department of Software Technology
Faculty EEMCS, Delft University of Technology
Delft, the Netherlands
www.ewi.tudelft.nl

Is cleaner greener? On the energy impact of refactored test smells

Author: Simon Biennier
Student id: 5551226

Abstract

Automated testing is essential for software reliability, yet test code frequently contains test smells that degrade maintainability. Prior research has largely examined these issues through software-quality perspectives, leaving their environmental impact underexplored. This study investigates how refactoring test smells affects energy consumption, execution time, and test quality in Java JUnit suites.

We curated a dataset of open-source systems, detected smells with *tsDetect* and manually validated refactorable instances. For each instance, we applied literature-backed, smell-specific refactorings and measured energy with *EnergiBridge* under controlled conditions. The results show that energy effects are smell-specific. Removing *Ignored test* instances yields clear energy savings, whereas refactoring the *Lazy test (JUnit 5)* smell via `@ParameterizedTest` incurs substantial energy increases. Most other smells exhibit small or inconsistent changes. Interestingly, we found that changes in energy were also strongly coupled with changes in execution time, within our evaluation context (a controlled, CPU-bound, sequential JUnit setting).

Overall, this study extends test smell research into software sustainability and highlights trade-offs between maintainability and energy efficiency. It provides a reproducible measurement pipeline and empirical guidance on when refactoring test smells is likely to be energy-beneficial.

Thesis committee:

Chair:	Andy Zaidman, Faculty EEMCS, TU Delft
University supervisor:	Andy Zaidman, Faculty EEMCS, TU Delft
Committee member:	Przemysław Pawełczak, Faculty EEMCS, TU Delft

Contents

List of Figures

Chapter 1

Introduction

In modern software engineering, automated testing plays an important role in ensuring the reliability and correctness of complex systems [5, 38, 67]. Despite its importance, test code is often treated with less rigour than production code and does not always co-evolve alongside it [46, 93].

Much like *code smells*, as defined by Fowler [27], poorly designed or inadequately maintained test code manifests as *test smells*: symptoms of suboptimal design decisions that hinder the clarity, maintainability, and long-term evolution of software systems. Originally defined by van Deursen et al. [85], and later refined and extended by Meszaros [48], *test smells* have since become the subject of extensive research. In particular, several empirical studies have investigated their properties [83, 84], distribution [8, 10], and impact on maintainability, showing that tests with test smells are more change- and defect-prone [77], hinder test comprehension, and increase maintenance effort [10, 78].

Historically, however, the academic and industrial focus has remained strictly on these quality-related dimensions, such as prevalence, detectability, and maintainability. What has been largely overlooked is the environmental dimension of these design choices. Research on green software engineering shows that if we want software to have a positive environmental impact, we need to design and run systems with sustainability in mind [23].

Test smells may not only hinder the quality of unit test suites, but may also be computationally wasteful. One potential example is the *General fixture* smell, where a test class sets up more objects than its tests actually need, as shown in Listing 1.1. "Such setUps are harder to read and understand. Moreover, they may make tests run more slowly (because they do unnecessary work)" [85], which, in turn, may increase their energy footprint. Conversely, certain refactorings intended to improve code quality might inadvertently increase energy usage.

The empirical investigation presented in this study fills this gap by evaluating how refactorings that remove test smells affect the energy consumption and quality of test suites. We address this by evaluating the direct environmental, operational, and structural implications

```java
public class OrderTest {
    private Database db;
    private User user;
    private Order order;

    @Before
    public void setUp() {
        db = new Database("db");
        user = new User("alice");
        order = new Order(42);
    }

    @Test
    public void testTotalCalculation() {
        assertEquals(42, order.getTotal());
    }
}
```

Listing 1.1: Example of a *General fixture* smell. Here, the setup initialises extra objects (`db` and `user`) that are not used by the test method.

of these changes, guided by the following research questions:

RQ1: To what extent does refactoring test smells affect the energy consumption of JUnit test suites?

RQ2: What is the relationship between changes in energy consumption and changes in execution time following refactoring?

RQ3: To what extent do test smell refactorings preserve or improve functional and structural test quality?

To address these research questions, we conducted an empirical investigation analysing 12 base types of test smells from the *tsDetect* tool [60]. By splitting two of these base smells into separate sub-variants, we expanded our scope to 14 distinct test smell types. The evaluation was conducted across 8 open-source Java systems. In total, we curated and evaluated 40 refactorable instances across 10 distinct test smell types, as 4 of our 14 studied types did not yield any instances. Each instance then underwent systematic, literature-backed refactoring, and its energy consumption, execution time, and test quality were subsequently measured using *EnergiBridge* [69] under controlled hardware conditions. To ensure statistical reliability and mitigate runtime fluctuations, we executed 60 experimental runs per instance, split equally between the original and refactored versions.

The remainder of this study is organised as follows. Chapter 2 provides background information on test smells and discusses the related literature, while Chapter 3 outlines our research design, including the refactoring process and energy measurement setup. Subsequently, Chapter 4 presents and analyses the experimental results, followed by a discussion of their implications in Chapter 5. Finally, Chapter 6 concludes the thesis and highlights directions for future work.

Chapter 2

Background and related work

In this chapter, we provide background information on test smells and discuss the related literature. In reviewing the related literature, we focus on work dealing with (i) test smells, (ii) test refactoring, and (iii) software sustainability.

2.1 Smells

As systems evolve, changes can introduce inconsistencies in the design and gradually degrade the structure of the software, making further evolution more difficult [62]. From this perspective, *code smells* can be seen as symptoms of these underlying design problems. More specifically, Fowler [28] defines a *code smell* (first coined by Beck [11]) as "a surface indication that usually corresponds to a deeper problem in the system['s design]" [28]. In this sense, code smells can act as indicators of issues that contribute to the accumulation of technical debt [84]. Abbes et al. [1] showed that combinations of *code smells* hinder program comprehension, whereas Khomh et al. [39] showed that "classes participating in [these] antipatterns are more change- and fault-prone than others". To this end, van Deursen et al. [85] similarly defined *test smells*, which indicated (potentially) poorly designed or inadequately maintained test code, and were later refined and extended by Meszaros [48].

Prior work has extensively investigated the nature, impact, and management of these *test smells*. Tufano et al. [83, 84] analyse their characteristics and evolution, showing that developers rarely perceive test smells as design issues, that most are introduced at test class creation rather than during later evolution, and that they tend to survive for long periods; they also report associations between certain test smells and production code smells, suggesting intertwined design problems.

Empirical studies by Bavota et al. [8, 10] demonstrated that test smells are widespread across both industrial and open-source systems, hinder test comprehension and increase maintenance effort. Focusing on quality, Spadini et al. [77] found that smelly tests are more change- and defect-prone, and that production code exercised by such tests is also more likely to contain defects.

Motivated by the need to manage this form of technical debt, automated detection tools have evolved significantly since the mid-2000s, advancing from early framework-specific

heuristics to sophisticated modern analysis platforms. Early efforts targeted niche environments, such as *TRex* [7] for TTCN-3 suites, *TestLint* [66] for Smalltalk systems, and the visually interactive *TestQ* [14] for C++.

Research then shifted towards Java, the most prominent ecosystem for smell detection [3]. Bavota et al. [8, 10] introduced a tool detecting nine smell types, though, due to its prioritisation of recall, it often necessitated manual verification. Specialised detection also emerged, such as *TestHound* [29] and its successor *TestEvoHound* [30], which focused exclusively on the evolution of test fixture code and recommended specific refactorings to improve fixture maintainability.

To address these traditional rule-based limitations, recent literature has branched out into alternative paradigms, such as machine learning and multi-language support. Machine learning models can outperform certain heuristics, but they struggle with overall effectiveness, often failing to exceed an average F-measure of 51% [65]. Conversely, multi-language frameworks have also emerged to address ecosystem-specific limitations; for instance, *AromaDr* [75] provides language-independent detection across C#, Java, JavaScript, TypeScript, and Python. Similarly, *SniffML* proposes a unified strategy to detect test smells across different xUnit frameworks by converting code into standardised XML [44].

Despite these technological advancements, integrating detection into practical software engineering workflows remains a challenge. Only a minority of tools report detailed correctness evidence, and reporting practices are often inconsistent, which motivates stronger expectations around smell-level precision/recall reporting, bias-mitigation disclosure, and shared benchmark datasets for fair comparison [3, 52]. Furthermore, to bridge the gap to developer workflows, tools like *DARTS* [43] and *RAIDE* [70] offer semi-automated refactoring support within the Eclipse environment. However, fully automated refactoring remains an open research challenge, highlighting a clear need to transition the field from mere smell detection to safe, automated refactoring [3].

Consequently, state-of-the-art heuristic tools that pair high precision with rigorous, fine-grained structural evaluation remain the industry standard. Representative of this mature class of detectors are JNose [88], which detects 21 test smell types alongside their code coverage metrics, and tsDetect [60], which extracts 19 unique smell types with an average F-score of 96.5%. Spadini et al. [78] built upon this structural baseline by introducing severity-based thresholds to better align tool reporting with actual developer perception.

In this study, we rely on *tsDetect* for test smell detection. The tool defines a catalogue of *test smells* based on established definitions in the literature, which forms the foundation of our study. Table 2.1 presents the 19 test smells supported by *tsDetect*.

Table 2.1: Test smells reported in the literature.

Abbr.	Test smell	Description	Possible effects
AR	Assertion roulette	Several assertions with no explanation within the same test method.	If an assertion fails, it can be difficult to identify which specific assertion caused the failure.
CTL	Conditional test logic	A test method contains branches such as if/else statements, loops, or switches.	Different outcomes depending on branch coverage make tests harder to understand and maintain.
CI	Constructor initialisation	The constructor of the test class performs initialisation logic.	Setup becomes implicit and harder to track; failures during construction cause all tests to fail.
DT	Default test	A default auto-generated test class remains in the codebase.	Wastes execution time and contributes nothing to test assurance.
DA	Duplicate assert	A test checks the same condition multiple times.	Adds noise and may indicate copy-paste errors or unclear test intent.
ET	Eager test	A test method checks several methods of the tested object.	Leads to difficulties in test comprehension and maintenance.
EmT	Empty test	A test method has no statements or is entirely blank.	Wastes execution time and provides no assurance of software quality.
EH	Exception handling	A test manually catches exceptions instead of letting the framework handle them.	Masks failures and makes the test intent unclear.
GF	General fixture	A test-case fixture is too general and the test methods only access part of it.	Increases difficulty in test comprehension due to unnecessary setup.
IgT	Ignored test	A test method is annotated to be skipped (e.g. `@Ignore`).	May hide incomplete or failing behaviour, reducing overall test reliability.
LT	Lazy test	Several test methods check a method of the tested class using the same fixture.	Difficulties in maintaining consistency during test suite updates.
MNT	Magic number test	A test uses unexplained numeric literals in assertions.	The lack of context makes the test significantly harder to understand.

MG	Mystery guest	A test uses external resources, such as a file containing test data.	Difficulties in test comprehension due to external, unknown values.
RA	Redundant assert	A test includes assertions that are always true or always false.	Wastes execution time and contributes nothing to test assurance.
RP	Redundant print	A test prints diagnostic output to the console.	Clutters test output and can hide important failure information.
RO	Resource optimism	A test makes assumptions about the state or existence of external resources.	Results in non-deterministic outcomes depending on the external environment.
SE	Sensitive equality	The `toString` method is used within assertion statements.	Failures may occur if the string representation is changed, regardless of logic.
ST	Sleepy test	A test uses `Thread.sleep()` to wait for specific behaviour.	Leads to fragile tests and unexpected results as processing times vary by device.
UT	Unknown test	A test method contains no assertions or validation logic.	Provides no feedback value and may give a false sense of code coverage.

2.2 Test refactoring

Refactoring is defined as the process of improving code structure without changing external behaviour [27]. While its positive impact on design [49, 54], maintainability [4], readability [61, 73] and defect prevention [42] is well established for production code [9, 73], refactoring is equally important for test code [48, 51].

Recent empirical studies, however, paint a nuanced picture of how test smell refactoring occurs in practice. Developers often appear to address smells opportunistically rather than as a primary goal, with motivations such as cleaning up redundant code, improving readability, or enhancing efficiency [62]. This suggests that while developers are aware of (test) smells, refactoring effort is frequently driven by broader maintenance concerns, which may help explain why smell removal remains limited in many systems.

Santana et al. [71] show that test smells frequently co-occur and that refactoring actions may have heterogeneous outcomes: some refactorings eliminate multiple smells at once, others address only a single instance and some fail to remove any smell at all. Kim et al. [40, 41] find that approximately 83% of test smell removals are a by-product of feature maintenance or restructuring, while only a small fraction of smells are intentionally addressed by developers. In many cases, refactoring merely relocates smells rather than eliminating them. Similarly, Chen et al. [16] observe that only a subset of test smells are consistently removed, suggesting that many academically defined smells do not align with developers' perceptions of test-related technical debt. In contrast, surveys and contribution studies indicate that when refactoring strategies are made explicit, developers largely agree with and accept them, implicitly validating the refactorings proposed in the literature [76].

Recent work has increasingly focused on making test smell refactoring more actionable by providing explicit refactoring strategies. In particular, Martins et al. [47] present a comprehensive catalogue of refactorings for handling test smells in Java-based systems. When combined with the smell types and detection rules implemented in *tsDetect* [60], this catalogue provides us with a structured basis for selecting the refactorings applied in our study.

2.3 Software sustainability

Society's growing emphasis on environmental conservation has placed software at the centre of sustainability discussions. The information and communication technology (ICT) sector consumed about 4% of global electricity in the use stage, representing about 1.4% of global greenhouse gas (GHG) emissions in 2020 [26]. Consequently, the computing community has identified the reduction of carbon emissions and environmental impact as a primary objective [17].

A core challenge of software sustainability is its inherent invisibility; because software is so intangible, its environmental footprint is exceptionally difficult to quantify and observe. Yet, while software does not consume energy directly, it dictates the hardware utilisation that drives this power consumption [63]. In the words of Benjamin Ninassi, "we must view digital resources as finite. We must accept that we cannot do everything without limit. We need to set priorities, just as we do with biomass." [13]. This creates a need

to develop sustainable software, also referred to as 'green software'. This pillar aims to improve software energy efficiency to minimise environmental footprints, which in turn fosters positive outcomes across broader sustainability dimensions [57, 15, 23].

Sustainability is commonly operationalised through energy consumption and, by extension, carbon emissions, as software directly influences how efficiently hardware resources are utilised [56]. Accurate measurement is therefore a prerequisite for understanding, comparing and ultimately improving the environmental footprint of software systems.

A common misconception in energy analysis is the assumption that energy consumption behaves identically to execution time. While the fundamental equation

$$E\ [\mathrm{J}] = P\ [\mathrm{W}] \times t\ [\mathrm{s}]$$

suggests that reducing execution time inherently lowers energy consumption, this assumes that power remains constant. In practice, power demand is dynamic and can fluctuate during execution. Consequently, empirical findings on this relationship are divided: while some work suggests that energy and time are strongly coupled [91], other studies have observed instances where the two metrics diverge [64, 82].

Existing approaches to measuring energy consumption can be broadly classified into three categories: (i) hardware-based, (ii) software-based, and (iii) estimation techniques [56]. Hardware power monitors provide highly accurate measurements but require specialised equipment and intrusive setups, limiting their scalability and practicality. Software-based energy profilers, often built on processor interfaces such as RAPL, offer a more lightweight alternative by estimating energy usage of components such as the CPU and memory, which consistently account for the majority of energy consumption in general-purpose software workloads [21]. Estimation approaches further reduce measurement overhead but trade accuracy for portability and ease of use.

In this study, we explicitly chose to measure energy consumption at the software level, using an energy profiler, rather than relying on external hardware meters. Although hardware-based alternatives offer high baseline accuracy, they are notoriously difficult to set up; they require custom physical connections to the device under study, as well as precise time-synchronisation frameworks between software execution and the power monitor to function reliably [21, 69]. Instead, we leverage software-level estimation, which provides a highly scalable and sufficiently precise alternative for our experimental setup. This choice is supported by empirical literature showing that validated software-reading mechanisms like Intel's RAPL (Running Average Power Limit) have been shown to be highly correlated with physical plug power, offering promising accuracy with negligible performance overhead [37], thereby serving as a robust tool for monitoring energy without the deployment of complex physical hardware meters.

2.3.1 Software energy consumption

Using different energy measurement approaches, a growing body of work has emerged on the environmental sustainability of software systems. For instance, Pereira et al. [58, 59] systematically evaluated and ranked the energy efficiency of various programming languages, while Pathak et al. [55] introduced fine-grained analysis to locate exactly where

energy is spent within an application. Moreover, research has explored architectural and deployment strategies aimed at limiting the environmental impact of software applications [31, 86]. Additional investigations have targeted post-deployment optimisations. These include the *Green mining* framework, which tracks software energy changes across successive commits [33], as well as techniques tailored specifically to reduce the power consumption of Android environments [24]. More recently, researchers have focused on minimising software testing overheads directly, proposing algorithmic frameworks for 'energy-aware' test case prioritisation [87].

However, research focusing specifically on the energy consumption of automated software testing remains in its early stages. Among the foundational studies, Cruz and Abreu [22] compared the energy consumption of different mobile testing frameworks and showed that testing tools themselves introduce distinct resource overheads. More recently, Zaidman [92] established an explicit baseline for test execution footprints, empirically demonstrating that testing a single open-source Java system can consume up to 117 kWh annually. Building on this work, Arntzenius et al. [6] profiled continuous integration (CI) workflows across Java systems running Maven and Gradle. Their large-scale analysis revealed that certain CI-intensive systems reach annual footprints of hundreds of kilowatt-hours; an expenditure comparable to a quarter of an average EU household's electricity usage.

While existing research measures macro-level CI infrastructure builds or relies on platform-level configurations, this study focuses directly on the micro-level structural architecture of the underlying test suite code. However, investigating this internal code space is historically challenging, as the historical diversity of platforms, operating systems, and processor architectures has hindered reproducible and comparable energy measurements [21]. To overcome these barriers and rigorously evaluate how internal test smells, and their subsequent structural refactoring, impact physical energy consumption, this study adopts *EnergiBridge*, a cross-platform energy measurement tool that supports multiple operating systems across different architectures [69].

Chapter 3

Research design

This chapter describes the research design of this study, including the selection of test smells, dataset curation, refactoring approach, and measurement process. This study's code and datasets can be found in the replication package [12].

3.1 Test smell selection

In this study, we leveraged the widespread adoption of Java and JUnit in both industry and open-source systems and, perhaps more importantly, the mature tooling available in their ecosystem. While alternative test smell detection tools such as *AromaDr* [75] and *SniffML* [44] were evaluated, they were ultimately found to be less suitable for this study due to their limited smell coverage and the difficulty of automating them at the scale required for our analysis. We instead selected *tsDetect* [60] as our primary detection engine, as it represents the most mature and widely adopted solution in the field [3]. This choice was driven by its ability to balance comprehensive coverage of 19 distinct smell types with a research-validated average F-score of 96.5%, whereas alternative frameworks often report lower accuracy scores or, in some cases, provide no empirical data at all [3].

From the 19 distinct smells that *tsDetect* is able to detect, we singled out 12 base test smells that we hypothesised may influence energy consumption. Although *tsDetect* only identifies these base smells, we manually inspected and partitioned two of them into distinct sub-types, resulting in a total of 14 evaluated smells. Specifically, *Lazy test* was split into *JUnit 4* and *JUnit 5* variants, because *JUnit 5* allows for clean refactoring via the `@ParameterizedTest` annotation, while *JUnit 4* often requires a manual data-driven approach to achieve the same result. Similarly, *Empty test* was divided into empty test methods and empty fixtures (`setUp`/`tearDown`). We decided to distinguish these base smells into sub-types, as they are fundamentally different in nature and therefore require distinct refactoring strategies. Table 3.1 provides an overview of the selected smells in black, the non-selected smells in grey, and a brief explanation of why each was included or excluded.

Table 3.1: Selected test smells, with their refactoring strategies and selection explanations.

Test smell	**Refactoring strategy**	**Selection explanation**
Assertion roulette	Add assertion message [85]	Considered largely obsolete in modern JUnit suites; adding assertion messages is unlikely to materially affect energy consumption.
Conditional test logic	Extract test [48]	Refactoring control-heavy tests into smaller tests may add setup/helper overhead, potentially increasing energy consumption.
Constructor initialisation	Move logic to `setUp()` [60]	Moving constructor logic to setup methods is expected to have a near-neutral energy effect and limited relevance in this study.
Default test	Remove test [60]	Removing placeholder tests is expected to reduce unnecessary execution overhead and therefore lower energy use.
Duplicate assert	Parameterise test [76]	Eliminating repeated assertions can reduce redundant computation, even if the expected energy reduction is small.
Eager test	Extract test [47, 85]	Splitting broad tests into focused ones can increase invocation and fixture overhead, making measurable energy impact plausible.
Empty test	Remove test [47]	Removing non-contributing test methods should reduce execution overhead and thus energy consumption.
Empty test (fixture)	Remove fixture [47]	Removing empty fixture methods should reduce setup/teardown overhead and thus energy consumption.
Exception handling	Use `assertThrows` [47, 76]	Replacing manual exception handling with assertion-based alternatives is expected to have little to no direct energy impact.
General fixture	Inline fixture [85]	Unnecessary shared fixture setup can increase setup/teardown work and localising fixtures may reduce this overhead.
Ignored test	Remove test [47]	Removing ignored tests may simplify test discovery and suite-management overhead, potentially reducing total energy use.
Lazy test (JUnit 4)	Parameterise test using data-driven approach [76]	Refactoring via parameterisation changes the execution profile and therefore warrants energy analysis.

Lazy test (JUnit 5)	Parameterise test using `@ParameterizedTest` [76]	Refactoring via parameterisation changes the execution profile and therefore warrants energy analysis.
Magic number test	Introduce named constants [60]	Replacing literals with named constants mainly improves readability and is not expected to materially affect runtime energy consumption.
Mystery guest	Inline or set up external resource [85]	Reducing external resource dependencies may lower I/O and setup overhead, potentially improving energy efficiency.
Redundant assert	Remove assertion [60]	Removing unnecessary assertions can slightly reduce execution work and therefore energy use.
Redundant print	Remove print [47]	Console output introduces avoidable I/O overhead that contributes directly to energy consumption.
Resource optimism	Set up external resource [85]	Primarily a correctness/reliability issue; fixing it changes invalid behaviour rather than isolating energy impact cleanly.
Sensitive equality	Introduce equality method [85]	Replacing sensitive, string-based equality checks is expected to have negligible impact on execution energy.
Sleepy test	Code addition [47]	Although potentially high-impact for energy, it often reflects deeper synchronisation problems that are hard to refactor consistently.
Unknown test	Add assertion [60]	Adding meaningful assertions may increase test execution time and energy consumption.

Table 3.2: Systems used in this study.

	Production code				Test code			
System	**Version**	**Classes**	**KLOC**	**Size (MB)**	**JUnit version**	**Classes**	**Methods**	**KLOC**
commons-codec	1.21.0	98	26	2.5	5.7.0	85	885	16
commons-io	2.22.0	206	61	6.1	5.14.1	17	2,379	39
commons-lang	3.21.0	187	100	11.1	5.14.2	11	4,427	66
commons-math	4.0	756	140	11.8	5.14.2	297	2,864	66
commons-numbers	1.3	551	123	8.5	5.13.4	247	839	89
gson	2.13.3	158	43	2.1	4.13.2	101	1,484	23
jfreechart	2.0.0	829	133	12.2	5.12.2	350	2,309	40
joda-time	2.14.0	88	312	8.2	4.13.2	135	4,329	55
All systems		**2,873**	**938**	**62.5**		**1,243**	**19,516**	**394**

3.2 Dataset

We attempted to use existing datasets from prior research; however, most of the systems investigated in that research were found to be outdated or could not be built.

Consequently, we opted to curate a new dataset of systems that were

(i) open-source,
(ii) written in Java,
(iii) included JUnit tests, and
(iv) contained less than 15 MB of source code.

This 15 MB constraint was chosen to ensure build feasibility. We initially attempted to reuse existing datasets and replication packages from prior studies, such as Panichella et al. [53] and Spadini et al. [78], but encountered significant difficulties because the included systems either failed to compile or demanded excessive build times on our local hardware. Compiling open-source software out of the box is notoriously difficult; studies show that only 41–47% of Java systems build successfully without manual intervention [32, 38, 81]. Consequently, capping the system size allowed us to quickly troubleshoot builds and significantly reduce compilation time during testing.

We used the *Defects4J* [35] dataset to select systems that met these requirements, then cloned the systems from GitHub and verified them by building and testing with Maven. Finally, *tsDetect* was executed using the sensitivity thresholds proposed by Spadini et al. [78] (`-t spadini`) to confirm the presence of at least one selected test smell in each system. Table 3.2 shows the systems that were included in this thesis, along with their respective sizes and test suite characteristics.

The next step was to manually validate the detected instances and retain only those that were refactorable. In practice, this step removed a substantial number of tool-reported instances. Many detections corresponded to tests that technically matched a smell rule but were semantically coherent and maintainable in context.

```java
public void testHalfdayNames() {
  DateTimeFormatter printer = DateTimeFormat.forPattern("a");
  for (int i=0; i<ZONES.length; i++) {
    Chronology chrono = ISOChronology.getInstance(ZONES[i]);
    MutableDateTime mdt = new MutableDateTime(2004, .., chrono);
    for (int hour=0; hour<24; hour++) {
      mdt.setHourOfDay(hour);
      int halfday = mdt.get(chrono.halfdayOfDay());
      String halfdayText = printer.print(mdt);
      assertEquals(HALFDAYS[halfday], halfdayText);
    }
  }
}
```

Listing 3.1: Example of *Conditional test logic* detection excluded from the dataset after manual review.

This observation is consistent with prior work questioning the practical precision of rule-based smell detection. In particular, Panichella et al. [53] report that many smell instances in manually written tests are false positives with limited relation to real maintainability concerns. Chen et al. [16] similarly show a misalignment between academically defined smells and developer-perceived quality problems. Following this evidence, we considered a refactoring feasible based on the criteria below.

(i) a documented refactoring strategy existed in established literature (see Table 3.1),
(ii) the refactoring would not decrease the test's internal quality (readability or maintainability), as assessed through manual review, and
(iii) the refactoring could be applied without changing the behaviour of the test, i.e., the test would preserve the same pass rate and would not decrease coverage or mutation scores.

One representative exclusion involved *Conditional test logic* detection in a loop-based test that validates half-day labels across all hours and time zones (see Listing 3.1). Although flagged as *Conditional test logic* due to iteration, this test is a domain-sweep invariant test rather than a simple branch-heavy unit test: it systematically checks half-day naming correctness across all 24 hours and multiple time zones (and similar patterns are used for month and weekday name validation). Flattening this logic into separate tests would produce hundreds of near-duplicate methods and reduce readability. We therefore classified this instance as non-refactorable for the purposes of this study.

A similar issue appeared for *Duplicate assert*. Many instances were flagged because they repeat the same assertion signature, i.e., same method and argument positions, while the asserted value changes after a deliberate state transition. Listing 3.2 shows an example. From a purely syntactic perspective, and thus to a heuristic-based detection tool, this looks like a *Duplicate assert*. Semantically, however, each assertion validates a different program state sequentially, so we excluded such instances.

```java
@Test
public void testEqualsNonEmptyObject() {
  JsonObject a = new JsonObject();
  JsonObject b = new JsonObject();

  new EqualsTester().addEqualityGroup(a).testEquals();

  a.add("foo", new JsonObject());
  assertThat(a.equals(b)).isFalse();
  assertThat(b.equals(a)).isFalse();

  b.add("foo", new JsonObject());
  MoreAsserts.assertEqualsAndHashCode(a, b);

  a.add("bar", new JsonObject());
  assertThat(a.equals(b)).isFalse();
  assertThat(b.equals(a)).isFalse();

  b.add("bar", JsonNull.INSTANCE);
  assertThat(a.equals(b)).isFalse();
  assertThat(b.equals(a)).isFalse();
}
```

Listing 3.2: Example of a *Duplicate assert* detection excluded. Repeated assertions validate different states in a sequential scenario.

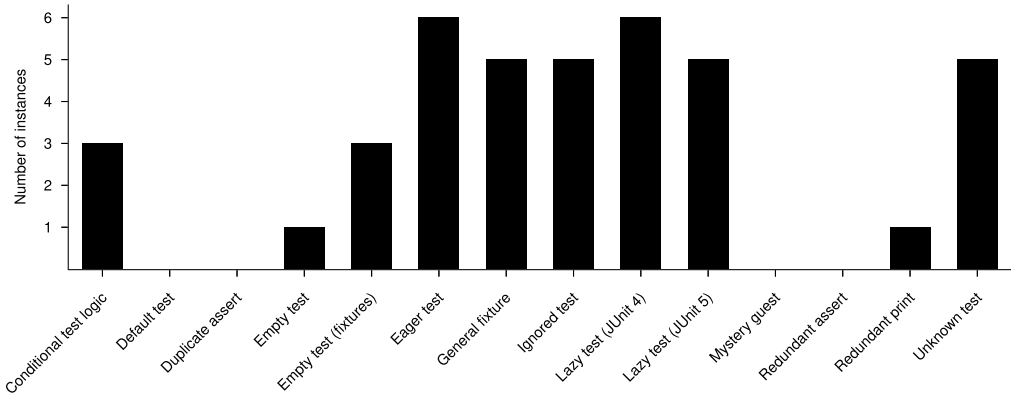

Figure 3.1: Distribution of test smell instances by type ($N = 40$).

Our target was to obtain at least five measurable instances per selected smell. In some cases this target was straightforward to meet, whereas for others, such as *Redundant assert*, it proved substantially more challenging. Ultimately, we retained a total of 40 instances across 10 distinct test smell types. The remaining 4 of our 14 studied test smells, *Default test*, *Duplicate assert*, *Mystery guest*, and *Redundant assert*, yielded no instances. Figure 3.1 and Table 3.3 provide an overview.

Table 3.3: List of all instances.

Instance	Smell	Explanation	Refactoring
1	General fixture	A variable is initialised in the `setUp` method for every test case, despite only being used in 17 out of 55 tests.	Inline fixture.
2	Unknown test	No assertions are performed; verifies only that the execution completes without throwing an exception.	Add assert statements.
3	Lazy test (JUnit 5)	Monolithic test cases perform many similar assertions to verify logic across different primitive types.	Parameterise test using `@ParameterizedTest`.
4	Ignored test	Contains `@Disabled` tests.	Remove `@Disabled` tests.
5	Ignored test	Contains `@Disabled` tests.	Remove `@Disabled` tests.
6	Ignored test	Contains `@Disabled` tests.	Remove `@Disabled` tests.
7	Ignored test	Contains `@Disabled` tests.	Remove `@Disabled` tests.
8	Ignored test	Contains `@Disabled` tests.	Remove `@Disabled` tests.
9	General fixture	A full suite of comparators (e.g., `cMillis` through `cYear`) is initialised in the `setUp` method for every execution, despite the majority of test cases requiring only a single instance.	Inline fixtures.
10	General fixture	Two variables are initialised in the `setUp` method for every test case, despite only being used in 2 out of 8 tests.	Inline fixtures.
11	Empty test (fixtures)	Empty `setUp` and `tearDown`.	Remove fixtures.
12	Empty test (fixtures)	Empty `setUp` and `tearDown`.	Remove fixtures.
13	Unknown test	No assertions are performed; verifies only that the execution completes without throwing an exception.	Add assert statements.
14	Empty test (fixtures)	Empty `setUp` and `tearDown`.	Remove fixtures.
15	Unknown test	No assertions are performed; verifies only that the execution completes without throwing an exception.	Add assert statements.
16	Unknown test	No assertions are performed; verifies only that the execution completes without throwing an exception.	Add assert statements.
17	Lazy test (JUnit 4)	Repetitive tests verify different date patterns using identical flows.	Parameterise test using data-driven loop.
18	Eager test	Groups independent checks of production methods.	Extract tests.
19	Eager test	Groups independent checks of production methods.	Extract tests.
20	Eager test	Groups independent checks of production methods.	Extract tests.
21	Lazy test (JUnit 4)	Repetitive tests verify different date formats using identical flows.	Parameterise test using data-driven loop.
22	Eager test	Contains ‘catch-all’ methods that test multiple behaviours (e.g., formatting, parsing and exception handling).	Extract tests.

23	Lazy test (JUnit 4)	Monolithic test cases perform many similar assertions to verify logic across different locales.	Parameterise test using data-driven loop.
24	Empty test	Empty tests.	Remove empty tests.
25	Redundant print	Test prints to the console (`System.out.println`).	Remove print statements.
26	General fixture	Resets the system's default TimeZone and Locale for every execution, despite these environment configurations only being required in 5 out of 11 tests.	Inline fixtures.
27	General fixture	A variable is initialised in the `setUp` method for every test case, despite only being used in 4 out of 9 tests.	Inline fixture.
28	Conditional test logic	Uses nested loops to dynamically calculate and verify expected array values.	Extract static array.
29	Conditional test logic	Uses loops for data generation and verification.	Extract loops into helper methods.
30	Lazy test (JUnit 4)	Redundant methods repeating identical logic for various inputs.	Parameterise test using data-driven loop.
31	Eager test	Lacks atomicity; groups independent checks of production methods.	Extract tests.
32	Conditional test logic	Uses nested loops to dynamically calculate and verify expected array values.	Extract static array.
33	Lazy test (JUnit 5)	Test performs many similar assertions to verify logic across different primitive types.	Parameterise test using `@ParameterizedTest`.
34	Lazy test (JUnit 5)	Test performs many similar assertions to verify logic across different primitive types.	Parameterise test using `@ParameterizedTest`.
35	Eager test	Groups independent checks of production methods.	Extract tests.
36	Lazy test (JUnit 5)	Test performs many similar assertions to verify logic across different primitive types.	Parameterise test using `@ParameterizedTest`.
37	Lazy test (JUnit 5)	Test performs many similar assertions to verify logic across different primitive types.	Parameterise test using `@ParameterizedTest`.
38	Unknown test	No assertions are performed; verifies only that the execution completes without throwing an exception.	Add assert statements.
39	Lazy test (JUnit 4)	Contains massive, redundant assertion blocks repeating the same length-comparison logic across various array types.	Parameterise test using data-driven loop.
40	Lazy test (JUnit 4)	Test repeats identical parsing logic across multiple different date formats.	Parameterise test using data-driven loop.

The retained instances were subsequently altered to isolate the identified smell by symmetrically removing unrelated tests from both the original and refactored versions. For example, in a test class exhibiting the *Eager test* smell, only the methods demonstrating that smell were retained. The unit of analysis is therefore an isolated smell instance rather than a complete system-level test suite, and the measured differences reflect the local effect of the smell-specific refactoring.

As refactoring is inherently subjective and test smells are only proxies for deeper design and implementation issues [16, 40, 41, 45], refactoring decisions were guided primarily by human-centric code quality (maintainability and readability) using refactoring strategies reported in the literature. Any change in energy consumption was treated as a secondary outcome rather than an optimisation target. To ensure functional correctness, i.e., that refactorings did not alter test behaviour, we verified all refactorings using two control metrics: *code coverage* (the percentage of production code executed by the test suite, measured with *JaCoCo* [50]) and *mutation score* (the percentage of artificially introduced faults, called *mutants*, detected by the test suite, measured with *PIT* [19]). Both tools were deployed using their out-of-the-box default configurations. Specifically, PIT was executed with its `DEFAULTS` mutation operator group.[1]

Although refactorings are assessed primarily through qualitative judgement of maintainability and readability, we also report the following metrics as supporting indicators:

Unit size, the size of the source code in the test class (measured in LOC).

Assertion density, the number of assertions relative to the size of the test code[2].

$$\text{Assertion density} = \frac{\#\text{assertions}}{LOC_{\text{test}}}$$

Listing 3.3 illustrates an example (instance 9, in simplified form) of an instance exhibiting the *General fixture* smell, in which the fixture setup is too general and the test methods only access part of it. Specifically, the `setUp` method sets up all comparators for milliseconds, seconds, minutes and hours, whereas the individual test methods each use only one comparator. The corresponding refactoring strategy (inlining fixtures) was applied, reducing the number of fixtures that are set up. This refactoring can be found in Listing 3.4 and mirrors the refactoring strategy applied across comparable instances.

The remaining refactorings are provided in the replication package [12].

3.3 Measurement setup

After refactoring, sustainability was evaluated with energy consumption as the primary metric. Measurements were collected with *EnergiBridge* [69], which reports telemetry at each

[1] `https://pitest.org/quickstart/mutators/`

[2] A custom script was used to perform a static count of explicit assertion calls within the test classes. Whilst this approach does not account for dynamic or external calls, it provides a consistent baseline for comparing the assertion density of the original and refactored versions.

```
@Override
public void setUp() {
    cSecond = DateTimeComparator.getInstance(..secondOfMinute());
    cMinute = DateTimeComparator.getInstance(..minuteOfHour());
    cHour = DateTimeComparator.getInstance(..hourOfDay());
}

@Override
protected void tearDown() {
    cSecond = null;
    cMinute = null;
    cHour = null;
}

public void testSecond() {
    aDateTime = getADate("1969-12-31T23:59:58");
    bDateTime = getADate("1969-12-31T23:50:59");
    assertEquals(-1, cSecond.compare(aDateTime, bDateTime));
}

public void testMinute() {
    aDateTime = getADate("1969-12-31T23:58:00");
    bDateTime = getADate("1969-12-31T23:59:00");
    assertEquals(-1, cMinute.compare(aDateTime, bDateTime));
}

public void testHour() {
    aDateTime = getADate("1969-12-31T22:00:00");
    bDateTime = getADate("1969-12-31T23:00:00");
    assertEquals(-1, cHour.compare(aDateTime, bDateTime));
}
```

Listing 3.3: Original code for instance 9, exhibiting the *General fixture* smell.

sampling interval (200ms by default). In this context, *PP0/Core* denotes energy consumed by CPU cores, *PP1/Uncore* denotes energy consumed by components close to the CPU (most commonly the integrated GPU), *DRAM* denotes memory energy and *Package/PKG* includes core and uncore domains [89]. For this study, the energy analysis is based on *Package* and *DRAM* counters, and all energy quantities are measured and reported in *Joules (J)*. To contextualise the energy results, we also measure *execution time* (in ms).

All experiments were conducted on a single, dedicated machine to eliminate cross-machine variability. Specifically, we used a Windows 11 (64-bit) machine equipped with an Intel Core i7 CPU @ 2.60 GHz and 16 GB DDR4 RAM, running Java 21.0.7 and Maven 3.9.12. To isolate hardware fluctuations, measurements were collected under strict, fixed conditions: the system was set to best-performance power mode, connected to a continuous plugged-in power source, adjusted to minimum fixed brightness, and disconnected from the network. To further reduce background noise and ensure reproducibility, we adopted a *zen mode* as defined by Cruz [20], which involved closing all non-essential applications, disabling system notifications, and disconnecting unnecessary hardware peripherals such as

```
// Removed setUp and tearDown

public void testSecond() {
    cSecond = DateTimeComparator.getInstance(..secondOfMinute());
    aDateTime = getADate("1969-12-31T23:59:58");
    bDateTime = getADate("1969-12-31T23:50:59");
    assertEquals(-1, cSecond.compare(aDateTime, bDateTime));
}

public void testMinute() {
    cMinute = DateTimeComparator.getInstance(..minuteOfHour());
    aDateTime = getADate("1969-12-31T23:58:00");
    bDateTime = getADate("1969-12-31T23:59:00");
    assertEquals(-1, cMinute.compare(aDateTime, bDateTime));
}

public void testHour() {
    cHour = DateTimeComparator.getInstance(..hourOfDay());
    aDateTime = getADate("1969-12-31T22:00:00");
    bDateTime = getADate("1969-12-31T23:00:00");
    assertEquals(-1, cHour.compare(aDateTime, bDateTime));
}
```

Listing 3.4: Refactored code for instance 9, with fixtures inlined.

external drives or displays. Furthermore, measurements were run at roughly 18.5 °C.

Once this environment was set up, we initiated a structured runtime execution and sampling pipeline. Before executing the target tests, we recorded a 30-second baseline trace to capture idle system energy consumption, followed by 12 warm-up runs lasting roughly 1 minute to stabilise the Java Virtual Machine (JVM). We enforced a 10-second pause between warm-up executions and an additional system-level stabilisation pause of 30 seconds between major phases. For the primary experiments, we configured 30 runs for both the original and refactored versions, yielding 60 scheduled runs per instance, with the execution order randomised prior to measurement to reduce temporal bias. For the *EnergiBridge* configuration, we retained the default sampling interval of 200 ms, as initial testing showed it produced the most stable readings. However, because this sampling window is relatively wide and may exceed the execution time of a single isolated test (if, for instance, a test runs for only 50–100 ms), we internally ran the test suite 10 times consecutively within each scheduled run to lengthen the measurement period and ensure the tool captured sufficient data. Finally, a 10-second cooldown was applied between consecutive scheduled runs to reduce tail-energy carry-over and limit collateral interference from tasks triggered by the previous execution [20].

Finally, to further minimise measurement overhead, the systems were built once in advance using `mvn -DskipTests package`, after which the energy measurements were executed by invoking the target tests directly through the JUnit runtime classpath (rather than through Maven test goals), which kept the execution pipeline as lean and accurate as possible.

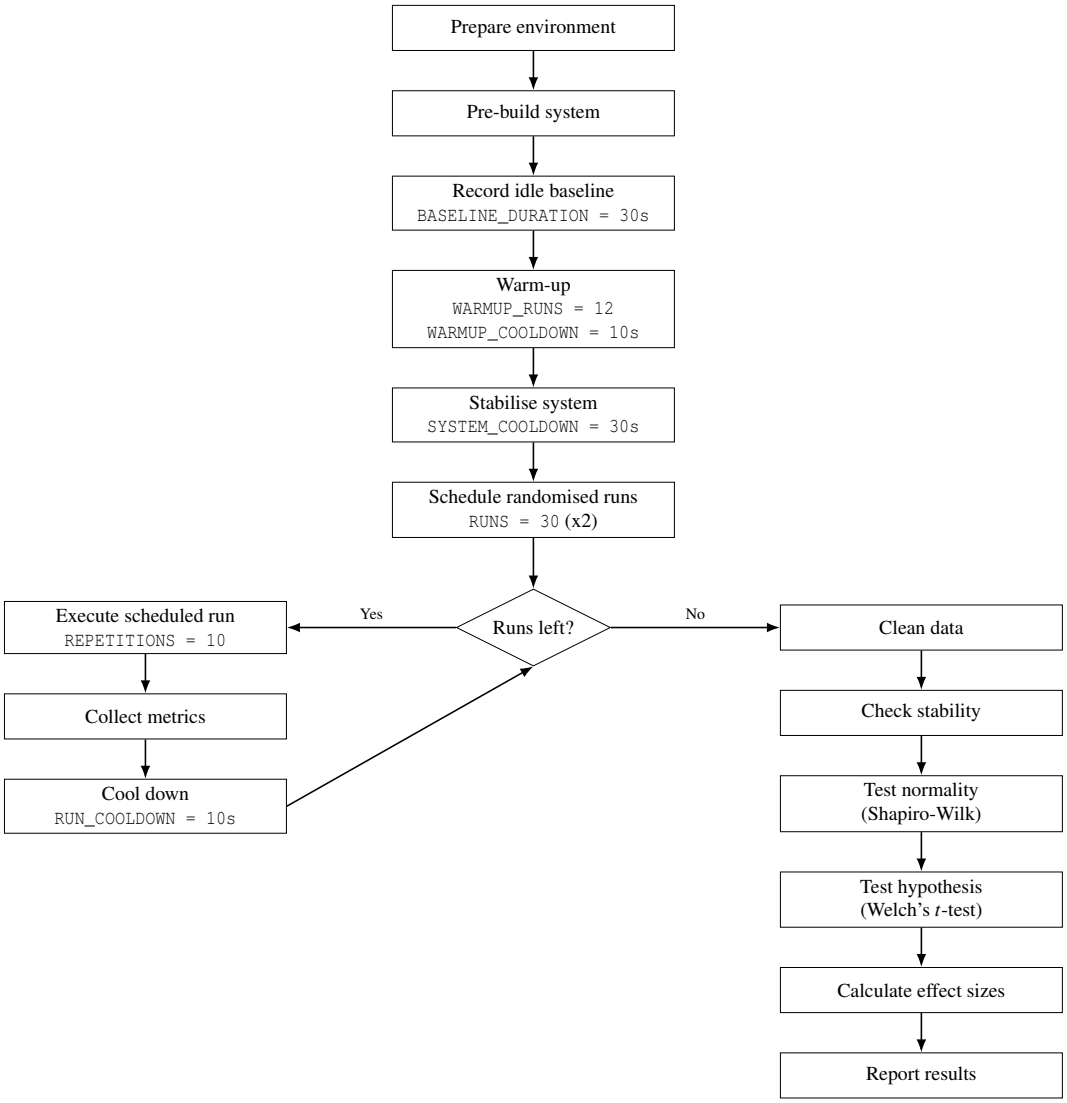

Figure 3.2: Measurement and analysis pipeline used in this study. Read top-to-bottom; the left branch loops over scheduled runs until none remain, then the right branch performs cleaning and analysis.

Figure 3.2 shows the pipeline for these measurements, together with their corresponding parameters. Through initial iterative testing, we determined the configuration that minimised execution runtime while ensuring the collected data remained stable and representative, according to the criteria outlined in the above section. This allowed us to maintain experimental standards without unnecessarily inflating the total compute time needed for the study.

3.4 Analysis

Because software energy measurements are inherently susceptible to environmental noise, such as unexpected background processes or system errors that may skew execution behaviour [20], we applied a two-step outlier treatment to eliminate unrepresentative data points before statistical analysis. We first removed negative energy values, which can occur due to occasional measurement overflow artefacts[3], and then applied a *modified Z-score* [34] filter to the remaining observations. We chose the modified Z-score rather than the standard Z-score because it is less sensitive to extreme values. The robust variant is based on the sample median and the median absolute deviation (MAD), which provides more stable outlier detection when a small number of runs deviate strongly from the rest [34].

We then assessed temporal stability by visualising all key metrics (including energy, CPU usage, memory usage and execution time) across run order. To support this analysis, we used a multi-metric dashboard that combines scatter plots with trend estimation and confidence intervals, allowing each metric to be classified as stable or drifting over time. Overall, this step provided a concise overview of measurement consistency prior to formal statistical testing. Examples of this workflow are shown for a single instance in Figure 3.3 and Figure 3.4. We initially wanted to use relative energy consumption, i.e., subtracting an idle baseline from the absolute recorded energy value in each session ($\Delta E = E_{\text{run}} - E_{\text{baseline}} = E_{\text{run}} - P_{\text{idle}} \times t$). However, these relative energy values proved to be too variable, likely due to background activity and timing noise, so the analysis in this study uses absolute measured energy values, which were much more stable overall.

After visual inspection, we assessed normality in the original and refactored samples using the *Shapiro-Wilk test* [74] as a data quality check. In this study, we treated samples with $p > 0.05$ on the Shapiro-Wilk test as approximately normal. We also evaluated whether the observed mean differences were statistically significant, and because variances may differ between the two samples, we used a two-sided parametric *Welch's t-test* [90]. We used a significance threshold of $\alpha = 0.05$, so Welch's t-test results with $p < 0.05$ were considered statistically significant, corresponding to 95% confidence. We preferred *Welch's test* in this context because it better controls Type I error under unequal variances or unbalanced sample sizes, while retaining comparable statistical power to other tests [68].

In addition to statistical significance, we quantified *effect sizes*, defined as "a quantitative reflection of the magnitude of some phenomenon that is used for the purpose of addressing a question of interest" [36], in both their standardised and unstandardised forms [25]. Specifically, we report the *mean difference* ($\Delta\mu$) and relative *percentage difference* ($\Delta\%$) as unstandardised effect sizes. To complement these, we calculate a standardised effect size using *Cohen's d* [18], which scales the effect against the data's variance. Cohen's d is interpreted as negligible when $d < 0.20$, small when $0.20 \leq d < 0.50$, moderate when $0.50 \leq d < 0.80$, large when $0.80 \leq d < 1.20$ and extremely large when $d \geq 1.20$ [18].

However, mathematical effect sizes, whether standardised or unstandardised, are fundamentally agnostic to the problem context. A mathematically 'large' statistical effect does not automatically equate to *substantive or practical significance* [36]. For instance, an op-

[3] https://github.com/tdurieux/EnergiBridge/issues/17

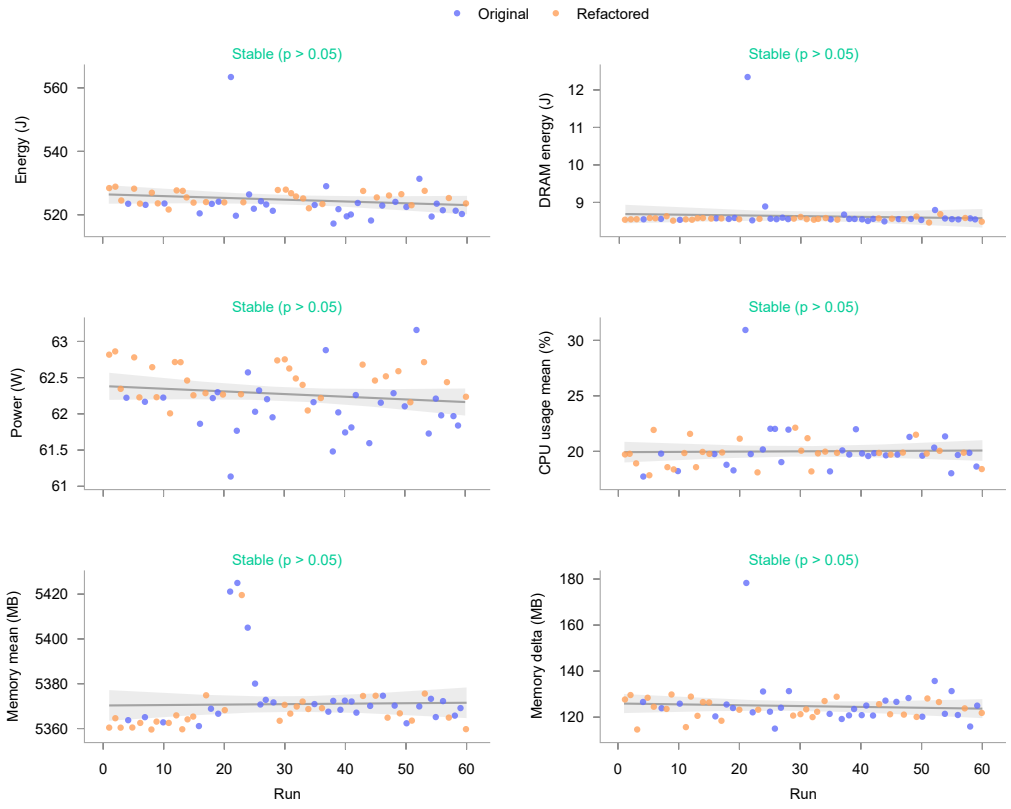

Figure 3.3: Temporal stability of metrics across runs. Points are ordered by run index; stable metrics appear as flat bands without drift. The plot shows that the key metrics remain temporally stable before hypothesis testing.

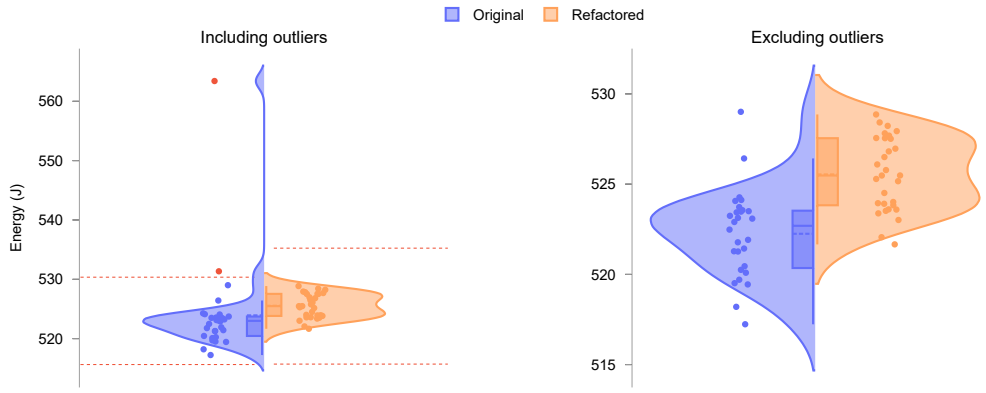

Figure 3.4: Energy distribution across runs, before and after outlier filtering. The left distribution shows raw measurements, the right shows values after removing negative readings and modified Z-score outliers. This allows us to identify outliers, variability, and the distributions of the energy measurements.

timisation that saves a Joule of energy across an entire day of intensive cloud software usage might yield a large statistical Cohen's *d* due to tight variance, yet offer negligible real-world insight [21]. Therefore, while these statistical effect metrics are necessary for rigorous analysis, we contextualise and critically analyse their true practical impact within chapters 4 and 5.

Finally, to assess the relationship between energy and execution time, we used *Spearman's rank correlation* (ρ) [79], because it is robust for noisy, non-linear data [72]. Following Schober et al. [72], correlations are considered negligible when $\rho < 0.10$, weak when $0.10 \leq \rho < 0.40$, moderate when $0.40 \leq \rho < 0.70$, strong when $0.70 \leq \rho < 0.90$ and extremely strong when $\rho \geq 0.90$. Identical intervals apply to negative values.

Chapter 4

Results and analysis

This chapter reports the empirical results together with their analysis, structured around the three research questions.

4.1 Measurement reliability

This section establishes the reliability of the measurement process before the main results are interpreted. Table 4.1 reports sample sizes, outlier counts, and normality checks to provide diagnostic evidence for measurement reliability. All metrics are reported for both original and refactored versions, and the differences are computed between the refactored and original versions ($RF - OR$). In this context, a positive difference indicates an increase in the metric of interest, and a negative difference indicates a reduction. All instances satisfy the normality assumption according to the Shapiro-Wilk tests ($p > 0.05$), and the outlier rate remains low (about 2–3%).

4.2 *RQ1: To what extent does refactoring test smells affect the energy consumption of JUnit test suites?*

In this chapter, we report results related to energy consumption by instance and smell type. To this end, Table 4.2 presents the absolute and percentage energy comparisons for each instance, alongside Welch's t-test outcomes and effect sizes.

A high-level view reveals that several instances within five smell categories exhibit statistically significant ($p < 0.05$) energy differences with large effects ($d > 0.8$), namely, *Eager test*, *General fixture*, *Ignored test*, *Lazy test (JUnit 4)*, and *Lazy test (JUnit 5)*. Based on these findings, combined with their prominence in existing literature and their high prevalence across our sampled systems, we treat these five smells as key smells in subsequent sections. Note that, although the *Empty test* smell also produced a statistically significant result, it was derived from a single instance, so we excluded it. These key smells, together with their median energy impacts, are shown in Table 4.3.

Interestingly, while Table 4.2 reports statistical significance and large effect sizes across several instances, only *Ignored test* and *Lazy test (JUnit 5)* exhibit practically meaningful

Table 4.1: Statistics and normality by test smell. Read *OR* and *RF* as original and refactored samples. Summary rows report the cumulative sum for sample sizes and absolute outlier counts, and the mean for outlier percentages. All rows satisfy normality.

		Sample size		Outliers				Shapiro-Wilk			
Test smell	**Instance**	OR	RF	OR	RF	Total	%	stat_{OR}	p_{OR}	stat_{RF}	p_{RF}
Conditional test logic	28	30	30	1	0	1	1.7	0.960	**0.327**	0.963	**0.375**
	29	30	30	1	2	3	5.0	0.969	**0.534**	0.971	**0.616**
	32	30	30	0	1	1	1.7	0.964	**0.381**	0.964	**0.416**
Total		**90**	**90**	**2**	**3**	**5**	**2.8**				
Eager test	18	30	30	2	0	2	3.3	0.961	**0.361**	0.949	**0.163**
	19	30	30	0	1	1	1.7	0.980	**0.818**	0.974	**0.671**
	20	30	30	0	0	0	0.0	0.954	**0.212**	0.979	**0.790**
	22	30	30	1	2	3	5.0	0.954	**0.226**	0.968	**0.525**
	31	30	30	1	0	1	1.7	0.975	**0.694**	0.978	**0.761**
	35	30	30	1	2	3	5.0	0.984	**0.922**	0.977	**0.776**
Total		**180**	**180**	**5**	**5**	**10**	**2.8**				
Empty test	24	30	30	1	0	1	1.7	0.946	**0.141**	0.977	**0.728**
Total		**30**	**30**	**1**	**0**	**1**	**1.7**				
Empty test (fixtures)	11	30	30	0	0	0	0.0	0.988	**0.981**	0.965	**0.404**
	12	30	30	0	0	0	0.0	0.975	**0.679**	0.973	**0.632**
	14	30	30	0	1	1	1.7	0.953	**0.201**	0.930	**0.056**
Total		**90**	**90**	**0**	**1**	**1**	**0.6**				
General fixture	1	30	30	0	2	2	3.3	0.983	**0.890**	0.977	**0.766**
	9	30	30	0	1	1	1.7	0.955	**0.227**	0.972	**0.604**
	10	30	30	0	0	0	0.0	0.964	**0.384**	0.986	**0.956**
	26	30	30	0	0	0	0.0	0.936	**0.072**	0.982	**0.873**
	27	30	30	1	0	1	1.7	0.940	**0.099**	0.972	**0.609**
Total		**150**	**150**	**1**	**3**	**4**	**1.3**				
Ignored test	4	30	30	1	4	5	8.3	0.974	**0.669**	0.926	**0.062**
	5	30	30	1	2	3	5.0	0.941	**0.107**	0.948	**0.178**
	6	30	30	0	0	0	0.0	0.961	**0.330**	0.964	**0.393**
	7	30	30	0	1	1	1.7	0.947	**0.141**	0.981	**0.874**
	8	30	30	1	0	1	1.7	0.958	**0.290**	0.961	**0.330**
Total		**150**	**150**	**3**	**7**	**10**	**3.3**				
Lazy test (JUnit 4)	17	30	30	0	1	1	1.7	0.966	**0.440**	0.958	**0.298**
	21	30	30	0	1	1	1.7	0.963	**0.375**	0.945	**0.132**
	23	30	30	0	2	2	3.3	0.960	**0.314**	0.971	**0.603**
	30	30	30	0	0	0	0.0	0.934	**0.063**	0.937	**0.075**
	39	30	30	0	1	1	1.7	0.964	**0.399**	0.977	**0.753**
	40	30	30	0	1	1	1.7	0.971	**0.558**	0.964	**0.419**
Total		**180**	**180**	**0**	**6**	**6**	**1.7**				
Lazy test (JUnit 5)	3	30	30	0	0	0	0.0	0.969	**0.505**	0.967	**0.465**
	33	30	30	0	0	0	0.0	0.984	**0.919**	0.979	**0.798**
	34	30	30	1	1	2	3.3	0.952	**0.203**	0.929	**0.052**
	36	30	30	1	0	1	1.7	0.975	**0.706**	0.980	**0.822**
	37	30	30	0	2	2	3.3	0.962	**0.357**	0.983	**0.915**
Total		**150**	**150**	**2**	**3**	**5**	**1.7**				
Redundant print	25	30	30	0	2	2	3.3	0.949	**0.162**	0.951	**0.213**
Total		**30**	**30**	**0**	**2**	**2**	**3.3**				
Unknown test	2	30	30	1	1	2	3.3	0.953	**0.223**	0.976	**0.728**
	13	30	30	0	0	0	0.0	0.957	**0.260**	0.984	**0.924**
	15	30	30	1	0	1	1.7	0.943	**0.121**	0.953	**0.204**
	16	30	30	0	3	3	5.0	0.971	**0.556**	0.954	**0.269**
	38	30	30	0	0	0	0.0	0.947	**0.139**	0.993	**> 0.999**
Total		**150**	**150**	**2**	**4**	**6**	**2.0**				
All instances		**1,200**	**1,200**	**16**	**34**	**50**	**2.1**				

Table 4.2: Energy metrics by test smell. Subscripts indicate original (OR) and refactored (RF) versions, and μ and σ denote mean and standard deviation, respectively. Bold values indicate statistical significance ($p < 0.05$) or large effects ($d > 0.8$). Summary rows report cumulative averages for means, pooled standard deviations for variances, and medians for percentage changes.

		Energy (J)				Welch's t-test		Effect size		
Test smell	**Instance**	μ_{OR}	σ_{OR}	μ_{RF}	σ_{RF}	t	p	$\Delta\mu$	$\Delta\%$	d
Conditional test logic	28	417.76	2.17	418.64	2.48	-1.460	0.150	0.88	0.21	0.38
	29	384.18	3.18	386.44	4.10	-2.319	**0.024**	2.26	0.59	0.62
	32	417.49	1.86	417.13	2.39	0.636	0.528	-0.36	-0.09	-0.17
Total		**406.48**	**2.47**	**407.41**	**3.09**			**0.93**	**0.21**	
Eager test	18	522.25	2.46	525.54	2.04	-5.521	**< 0.001**	3.29	0.63	**1.46**
	19	522.61	3.28	526.32	2.46	-4.920	**< 0.001**	3.71	0.71	**1.28**
	20	609.39	5.03	613.44	4.01	-3.453	**0.001**	4.05	0.67	**0.89**
	22	429.75	2.35	433.86	2.09	-6.992	**< 0.001**	4.11	0.96	**1.85**
	31	388.04	2.51	388.99	2.38	-1.487	0.143	0.95	0.24	0.39
	35	432.58	1.93	433.38	2.06	-1.509	0.137	0.80	0.18	0.40
Total		**484.10**	**3.10**	**486.92**	**2.60**			**2.82**	**0.65**	
Empty test	24	380.83	1.79	378.57	2.65	3.841	**< 0.001**	-2.25	-0.59	**-0.99**
Total		**380.83**	**1.79**	**378.57**	**2.65**			**-2.25**	**-0.59**	
Empty test (fixtures)	11	322.49	3.58	324.06	4.90	-1.414	0.163	1.57	0.49	0.37
	12	332.65	3.79	333.13	3.78	-0.484	0.630	0.47	0.14	0.12
	14	426.80	2.50	425.57	2.70	1.816	0.075	-1.23	-0.29	-0.47
Total		**360.65**	**3.34**	**360.92**	**3.90**			**0.27**	**0.14**	
General fixture	1	593.34	7.14	586.40	8.60	3.331	**0.002**	-6.94	-1.17	**-0.88**
	9	385.91	4.26	385.07	4.25	0.762	0.449	-0.84	-0.22	-0.20
	10	339.05	3.83	340.08	3.97	-1.026	0.309	1.03	0.30	0.27
	26	520.04	2.80	519.48	3.65	0.671	0.505	-0.56	-0.11	-0.17
	27	436.21	1.87	437.73	2.04	-2.993	**0.004**	1.52	0.35	0.78
Total		**454.91**	**4.36**	**453.75**	**5.00**			**-1.16**	**-0.11**	
Ignored test	4	469.75	3.94	357.69	3.81	107.167	**< 0.001**	-112.06	**-23.86**	**-28.89**
	5	393.28	4.81	345.89	2.04	48.696	**< 0.001**	-47.39	**-12.05**	**-12.74**
	6	656.72	4.80	412.90	2.51	246.558	**< 0.001**	-243.82	**-37.13**	**-63.66**
	7	491.62	2.90	413.45	2.55	110.168	**< 0.001**	-78.18	**-15.90**	**-28.63**
	8	450.20	2.32	412.56	1.70	70.813	**< 0.001**	-37.64	**-8.36**	**-18.54**
Total		**492.31**	**3.89**	**388.50**	**2.62**			**-103.82**	**-15.90**	
Lazy test (JUnit 4)	17	474.09	2.53	467.73	2.42	9.875	**< 0.001**	-6.36	-1.34	**-2.57**
	21	560.41	2.14	556.80	4.49	3.919	**< 0.001**	-3.61	-0.64	**-1.03**
	23	364.63	3.46	366.56	2.57	-2.426	**0.019**	1.93	0.53	0.63
	30	327.00	4.01	322.74	3.41	4.431	**< 0.001**	-4.26	-1.30	**-1.14**
	39	419.08	2.01	412.17	2.00	13.253	**< 0.001**	-6.92	-1.65	**-3.45**
	40	393.87	2.05	393.61	1.73	0.522	0.604	-0.26	-0.07	-0.14
Total		**423.18**	**2.81**	**419.94**	**2.92**			**-3.24**	**-0.97**	
Lazy test (JUnit 5)	3	360.50	9.45	585.00	20.72	-53.985	**< 0.001**	224.50	**62.27**	**13.94**
	33	366.90	3.88	392.58	3.87	-25.656	**< 0.001**	25.68	**7.00**	**6.62**
	34	369.61	3.08	449.06	4.05	-84.164	**< 0.001**	79.45	**21.50**	**22.10**
	36	417.45	1.73	485.08	1.96	-140.603	**< 0.001**	67.63	**16.20**	**36.54**
	37	444.57	2.26	532.96	3.94	-103.787	**< 0.001**	88.38	**19.88**	**27.76**
Total		**391.81**	**4.94**	**488.94**	**9.80**			**97.13**	**19.88**	
Redundant print	25	396.21	3.75	395.53	2.21	0.843	0.404	-0.68	-0.17	-0.22
Total		**396.21**	**3.75**	**395.53**	**2.21**			**-0.68**	**-0.17**	
Unknown test	2	303.47	8.76	304.69	8.34	-0.547	0.587	1.23	0.40	0.14
	13	393.02	1.80	393.10	2.05	-0.156	0.877	0.08	0.02	0.04
	15	329.12	4.34	329.32	5.17	-0.157	0.876	0.20	0.06	0.04
	16	397.07	2.16	396.41	2.24	1.130	0.264	-0.66	-0.17	-0.30
	38	422.26	2.65	421.57	2.47	1.037	0.304	-0.69	-0.16	-0.27
Total		**368.99**	**4.70**	**369.02**	**4.73**			**0.03**	**0.02**	
All instances		**426.55**	**3.80**	**425.53**	**4.82**			**-1.02**	**0.04**	

Table 4.3: Key test smells with their median energy changes (in percent). *Ignored test* and *Lazy test (JUnit 5)* show the largest effects.

Test smell	**Energy change (%)**
Eager test	+0.6
General fixture	-0.1
Ignored test	**-15.9**
Lazy test (JUnit 4)	-1.0
Lazy test (JUnit 5)	**+19.9**

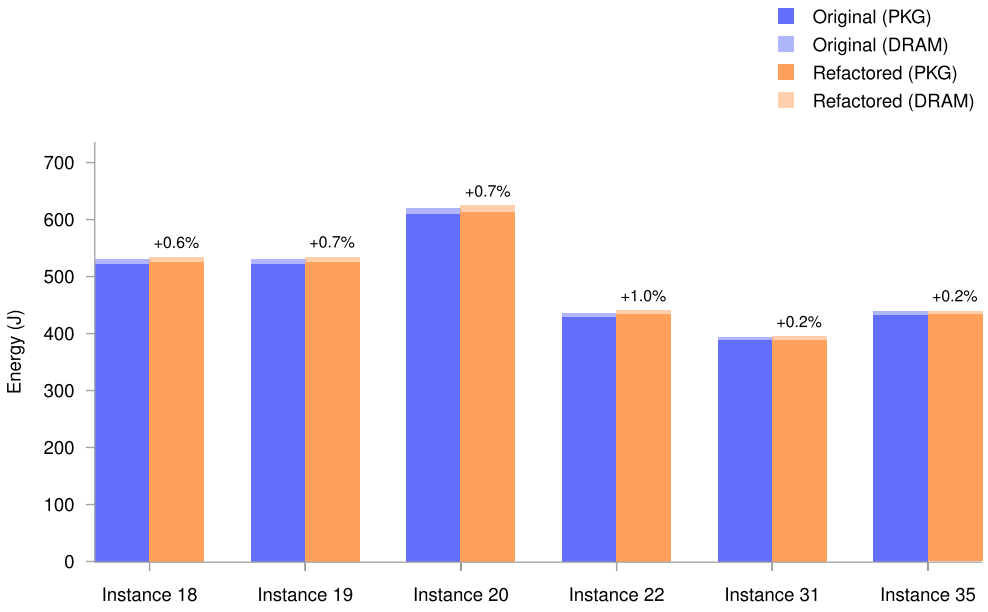

Figure 4.1: Energy differences for *Eager test* instances. Bars show package and DRAM energy consumption, with the instance-level percent energy change annotated. Changes are small, suggesting limited practical impact.

energy changes, yielding a 16% reduction and a 20% increase, respectively. These variations are illustrated via the bar charts in Figures 4.1–4.5, which visually reinforce these observations.

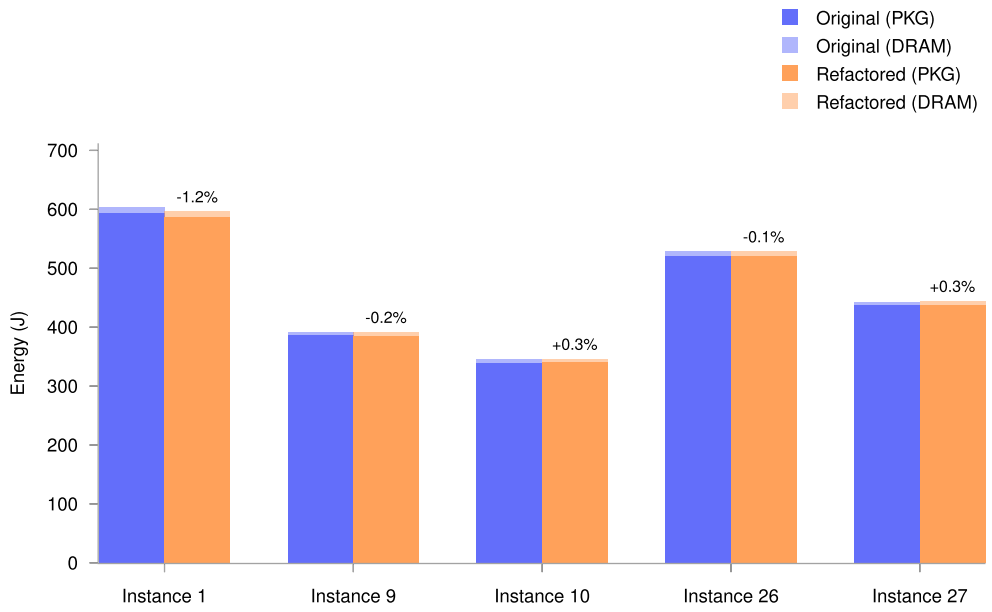

Figure 4.2: Energy differences for *General fixture* instances. Effects are mixed and small, suggesting limited practical impact.

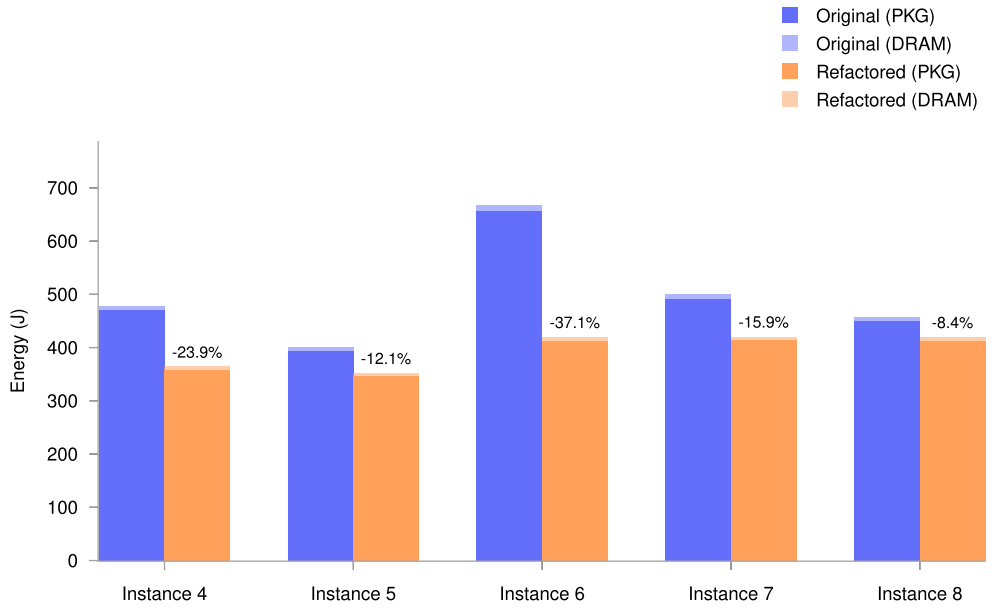

Figure 4.3: Energy differences for *Ignored test* instances. All instances show clear reductions, indicating a consistent energy saving.

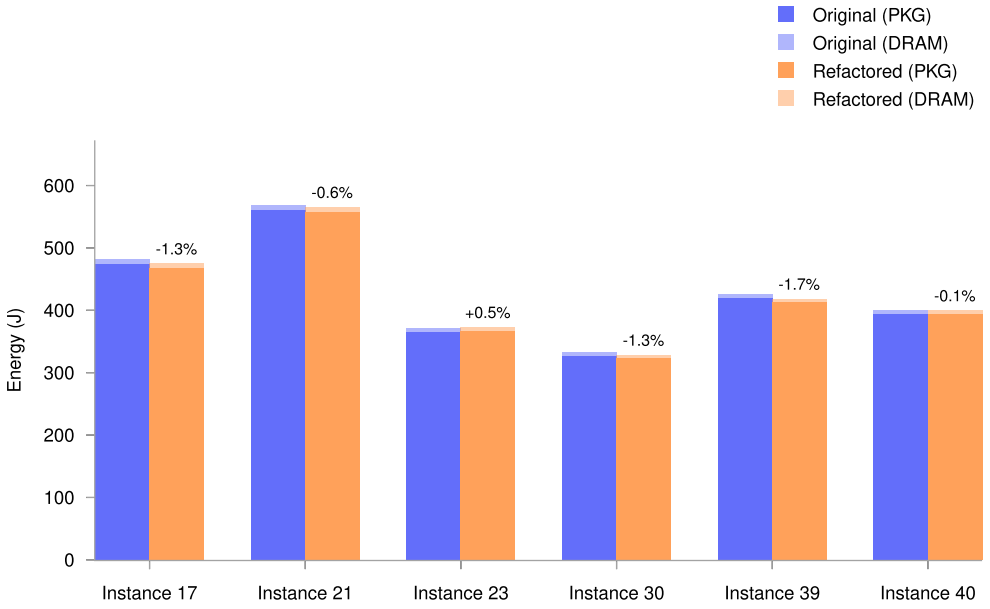

Figure 4.4: Energy differences for *Lazy test (JUnit 4)* instances. Changes are small and inconsistent, suggesting limited practical impact.

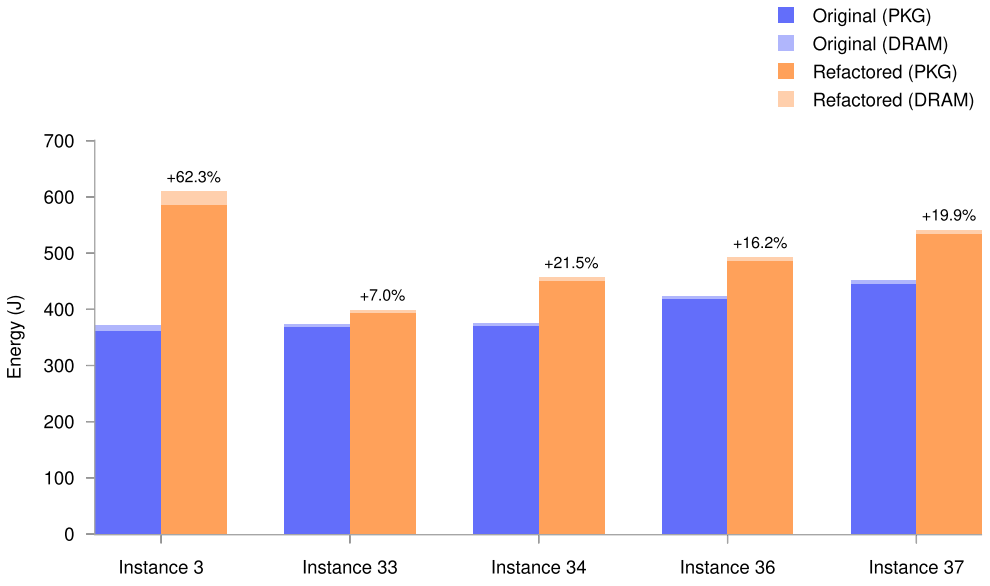

Figure 4.5: Energy differences for *Lazy test (JUnit 5)* instances. All instances increase energy substantially, indicating a systematic energy cost.

4.3 *RQ2: What is the relationship between changes in energy consumption and changes in execution time following refactoring?*

While the findings in the previous section demonstrate that refactoring specific test smells alters energy consumption, they do not establish whether these changes are independent of execution time. To understand the underlying drivers of these energy changes, this section evaluates the relationship between changes in energy consumption and changes in execution time following refactoring. We begin this investigation by evaluating our data through an initial visual inspection to see if changes in energy map intuitively onto changes in time. To do this, we present energy boxplots for both the original and refactored versions, together with their corresponding execution times plotted as discrete points. These can be found in Figures 4.6–4.10.

For several key smells, the absolute execution times before and after refactoring appear mostly identical. Visually, this suggests that refactoring alters the energy footprint while leaving execution time mostly unchanged. However, as demonstrated below, analysing these distributions purely by visual scale masks a much tighter, underlying statistical reality. In other words, small absolute shifts in time are visually compressed on these plots, yet they still co-move consistently with energy when expressed as relative differences. Notably, Table 4.4 summarises the median percentage differences ($\Delta\%$) for both energy and execution time alongside their overall Spearman rank correlations (ρ).

Table 4.4: Median changes and correlation between energy and execution time, for each test smell. Positive values indicate increases after refactoring, negative values indicate decreases, and bold values highlight notable percentage differences ($> 5\%$) and strong statistical relationships ($\rho > 0.70$). Most key smells, shown in bold, show strong correlations.

Test smell	Energy change (%)	Time change (%)	Correlation
Conditional test logic	+0.2	+0.1	0.47
Empty test	-0.6	+0.0	-0.32
Empty test (fixtures)	+0.1	-0.0	**0.86**
Eager test	+0.6	+0.0	**0.93**
General fixture	-0.1	-0.0	**0.90**
Ignored test	**-15.9**	**-10.9**	**0.93**
Lazy test (JUnit 4)	-1.0	-0.0	**0.78**
Lazy test (JUnit 5)	**+19.9**	**+21.5**	0.53
Redundant print	-0.2	-0.1	0.08
Unknown test	+0.0	+0.1	-0.30

As demonstrated by the correlations in Table 4.4, most of our key smells show strong correlations between energy and execution time. The sole exception is *Lazy test (JUnit 5)*, which exhibits a moderate correlation ($\rho = 0.53$), likely due to its unique setup overhead.

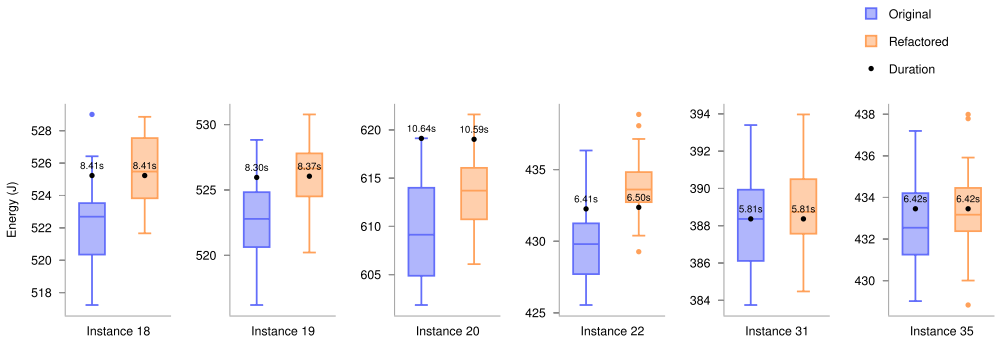

Figure 4.6: Energy boxplots for original and refactored versions, along with execution time data for *Eager test* instances. Energy changes are more visible than changes in execution time.

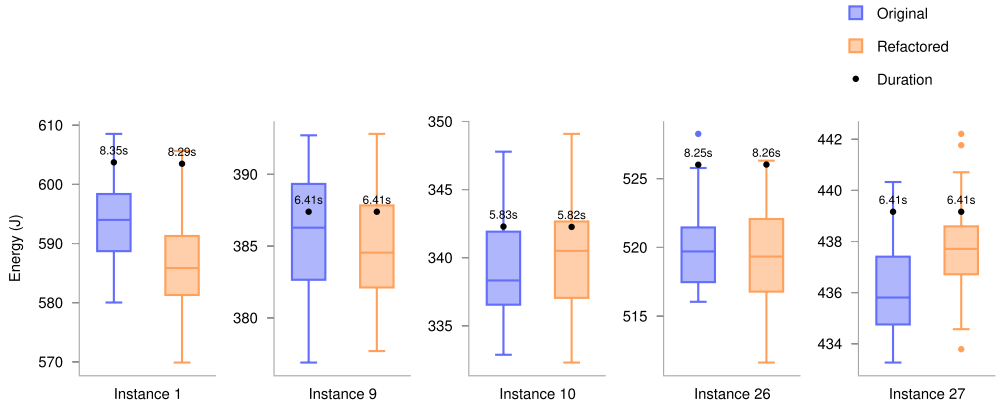

Figure 4.7: Energy boxplots for original and refactored versions, along with execution time data for *General fixture* instances. Energy changes are more visible than changes in execution time.

Furthermore, aggregating all instances using absolute energy and execution time values also yields a strong correlation ($\rho = 0.725, p < 0.001$). However, because absolute correlations are more susceptible to external factors, we now turn our focus to relative percentage differences.

Visually, these patterns can be found in Figure 4.11, a scatter plot displaying percentage differences ($\Delta\%$) in energy versus percentage differences in execution time. This visualisation reveals two clear patterns: first, as observed previously, visible energy changes are concentrated primarily in the highlighted smells (*Ignored test* and *Lazy test (JUnit 5)*), where all exceed 5%, while most other instances cluster near zero. Second, when energy changes are substantial, changes in time tend to move in the same direction, demonstrated by the (positive) linear trend. This is reflected in a strong overall correlation ($\rho = 0.765$, $p < 0.001$) and an even stronger relationship within highlighted smells ($\rho = 0.988$, $p < 0.001$). These

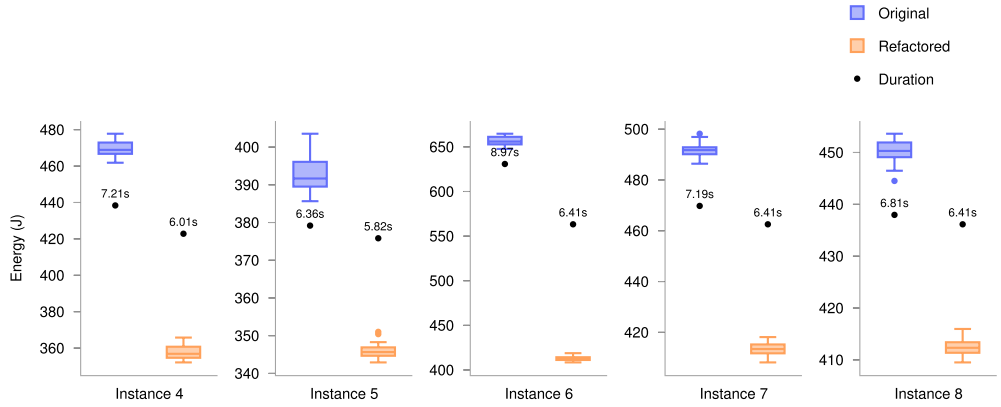

Figure 4.8: Energy boxplots for original and refactored versions, along with execution time data for *Ignored test* instances. Both energy and time changes are visible.

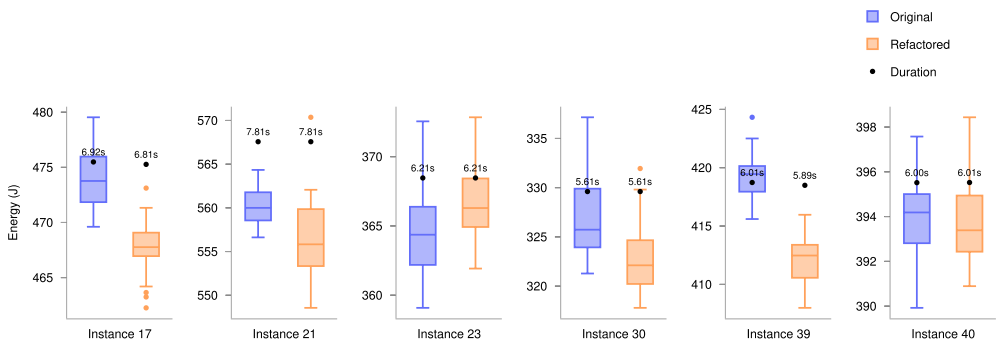

Figure 4.9: Energy boxplots for original and refactored versions, along with execution time data for *Lazy test (JUnit 4)* instances. Energy changes are more visible than changes in execution time.

findings collectively suggest that, within this context, energy and time seem to be tightly coupled, both at absolute and relative levels.

4.4 *RQ3: To what extent do test smell refactorings preserve or improve functional and structural test quality?*

Following the analysis of the statistical relationship between energy and execution time, it is crucial to evaluate how these refactorings impact the test suites themselves. A core requirement of any refactoring approach is that it must not compromise test suite integrity. Consequently, this section assesses whether these refactoring strategies preserve or improve functional and structural test quality. To do this, we use coverage and mutation scores, as well as structural metrics such as unit size and assertion density, as our primary indicators.

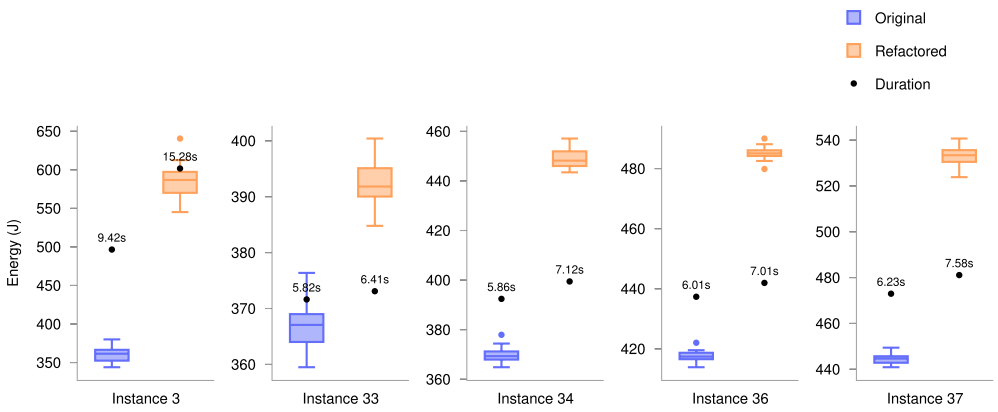

Figure 4.10: Energy boxplots for original and refactored versions, along with execution time data for *Lazy test (JUnit 5)* instances. Both energy and time changes are visible.

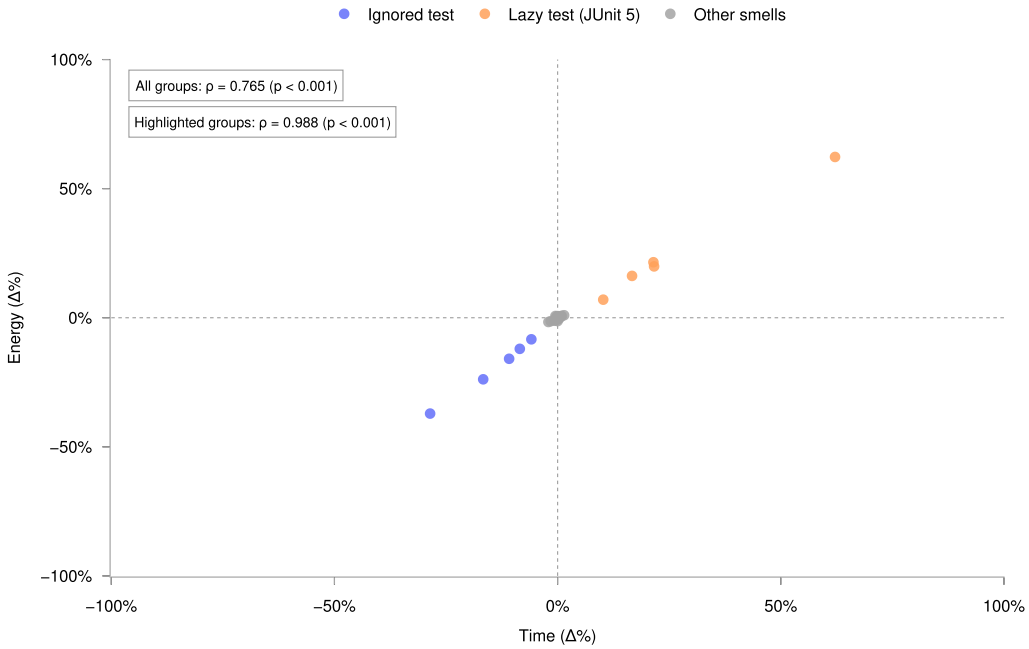

Figure 4.11: Scatter plot illustrating the relative change (percent) in energy consumption versus execution time across all instances, coloured by smell type. The clear linear trend and strong positive correlations demonstrate that energy savings are tightly coupled with execution time reductions, and vice versa.

Table 4.5 reports functional indicators: branches, coverage, and mutation scores. Table 4.6 reports structural indicators: unit size (LOC), number of assertions, and assertion density.

Crucially, coverage and mutation scores act largely as our control metrics, ensuring that the behavioural integrity of the test suites was maintained during refactoring. Hence, these metrics remain broadly stable across instances, demonstrating that the refactorings typically preserved or slightly improved test efficacy, and confirming a controlled experimental baseline. In contrast to these stable control metrics, the structural changes detailed in Table 4.6 are more pronounced, but also directly reflect the mechanics of each specific refactoring strategy. For instance, *Eager test* increases unit size (LOC) and assertion counts as eager tests are split and extracted into dedicated methods. Conversely, *General fixture* increases unit size only, reflecting the inlining of fixtures. The most dramatic reductions occur in *Ignored test*, which shows a substantial reduction in unit size as these ignored tests are removed. Similarly, both *Lazy test* variants show large decreases in unit size and assertion count, which serves as direct evidence that assertions were successfully parameterised into concise, data-driven tests with fewer explicit assertions. Finally, we observe that *Unknown test* increases unit size and assertion count as missing validation checks are added. Overall, while unit size and assertion count decrease across all instances combined, these aggregated totals should be interpreted cautiously because the underlying distributions differ markedly by smell type.

Table 4.5: Coverage and mutation metrics by test smell. Read *OR* and *RF* as original and refactored samples, and $\Delta = RF - OR$. Rows with changes between original and refactored versions ($\Delta \neq 0$) are marked in black, and unchanged rows are marked in grey.

		Branches				Code coverage		Mutants				Mutation score	
Test smell	**Instance**	OR	RF	Δ	Total	OR	RF	OR	RF	Δ	Total	OR	RF
Conditional test logic	28	201	214	+13	2,724	0.074	0.079	14	14	0	117	0.120	0.120
	29	80	80	0	2,724	0.029	0.029	52	52	0	109	0.477	0.477
	32	217	221	+4	2,724	0.080	0.081	16	16	0	117	0.137	0.137
Total		**498**	**515**	**+17**	**8,172**	**0.061**	**0.063**	**82**	**82**	**0**	**343**	**0.244**	**0.244**
Eager test	18	501	501	0	6,678	0.075	0.075	0	0	0	0	0.000	0.000
	19	501	501	0	6,678	0.075	0.075	39	39	0	42	0.929	0.929
	20	537	537	0	6,678	0.080	0.080	11	11	0	25	0.440	0.440
	22	718	718	0	6,678	0.108	0.108	6	6	0	117	0.051	0.051
	31	97	97	0	2,724	0.036	0.036	0	0	0	159	0.000	0.000
	35	412	413	+1	6,678	0.062	0.062	8	8	0	197	0.041	0.041
Total		**2,766**	**2,767**	**+1**	**36,114**	**0.073**	**0.073**	**64**	**64**	**0**	**540**	**0.243**	**0.243**
Empty test	24	16	16	0	11,746	0.001	0.001	4	4	0	16	0.250	0.250
Total		**16**	**16**	**0**	**11,746**	**0.001**	**0.001**	**4**	**4**	**0**	**16**	**0.250**	**0.250**
Empty test (fixtures)	11	131	131	0	6,678	0.020	0.020	7	7	0	7	1.000	1.000
	12	373	373	0	6,678	0.056	0.056	12	12	0	71	0.169	0.169
	14	535	535	0	6,678	0.080	0.080	0	0	0	150	0.000	0.000
Total		**1,039**	**1,039**	**0**	**20,034**	**0.052**	**0.052**	**19**	**19**	**0**	**228**	**0.390**	**0.390**
General fixture	1	425	425	0	9,687	0.044	0.044	130	130	0	142	0.915	0.915
	9	813	813	0	6,678	0.122	0.122	43	43	0	46	0.935	0.935
	10	330	330	0	6,678	0.049	0.049	3	3	0	3	1.000	1.000
	26	235	235	0	2,724	0.086	0.086	8	8	0	13	0.615	0.615
	27	475	475	0	2,724	0.174	0.174	41	41	0	117	0.350	0.350
Total		**2,278**	**2,278**	**0**	**28,491**	**0.095**	**0.095**	**225**	**225**	**0**	**321**	**0.763**	**0.763**
Ignored test	4	8	8	0	9,687	0.001	0.001	0	0	0	142	0.000	0.000
	5	7	7	0	9,687	0.001	0.001	0	0	0	142	0.000	0.000
	6	0	0	0	9,687	0.000	0.000	0	0	0	142	0.000	0.000
	7	0	0	0	9,687	0.000	0.000	0	0	0	142	0.000	0.000
	8	0	0	0	9,687	0.000	0.000	0	0	0	142	0.000	0.000
Total		**15**	**15**	**0**	**48,435**	**0.000**	**0.000**	**0**	**0**	**0**	**710**	**0.000**	**0.000**
Lazy test (JUnit 4)	17	750	751	+1	6,678	0.112	0.112	55	58	+3	381	0.144	0.152
	21	617	617	0	6,678	0.092	0.092	14	14	0	264	0.053	0.053
	23	413	413	0	6,678	0.062	0.062	8	8	0	197	0.041	0.041
	30	35	35	0	2,724	0.013	0.013	18	18	0	86	0.209	0.209
	39	22	22	0	9,687	0.002	0.002	21	21	0	1,862	0.011	0.011
	40	385	385	0	6,678	0.058	0.058	14	14	0	381	0.037	0.037
Total		**2,222**	**2,223**	**+1**	**39,123**	**0.057**	**0.057**	**130**	**133**	**+3**	**3,171**	**0.083**	**0.084**
Lazy test (JUnit 5)	3	302	302	0	9,687	0.031	0.031	0	0	0	142	0.000	0.000
	33	16	16	0	9,687	0.002	0.002	8	8	0	1,862	0.004	0.004
	34	76	76	0	9,687	0.008	0.008	83	83	0	1,862	0.045	0.045
	36	61	61	0	9,687	0.006	0.006	67	67	0	1,862	0.036	0.036
	37	54	54	0	9,687	0.006	0.006	63	63	0	1,862	0.034	0.034
Total		**509**	**509**	**0**	**48,435**	**0.011**	**0.011**	**221**	**221**	**0**	**7,590**	**0.024**	**0.024**
Redundant print	25	30	30	0	11,746	0.003	0.003	32	32	0	92	0.348	0.348
Total		**30**	**30**	**0**	**11,746**	**0.003**	**0.003**	**32**	**32**	**0**	**92**	**0.348**	**0.348**
Unknown test	2	0	0	0	9,687	0.000	0.000	0	0	0	142	0.000	0.000
	13	0	0	0	6,678	0.000	0.000	0	0	0	0	0.000	0.000
	15	192	204	+12	6,678	0.029	0.031	0	0	0	150	0.000	0.000
	16	20	20	0	6,678	0.003	0.003	5	5	0	182	0.027	0.027
	38	488	488	0	6,678	0.073	0.073	25	28	+3	221	0.113	0.127
Total		**700**	**712**	**+12**	**36,399**	**0.021**	**0.021**	**30**	**33**	**+3**	**695**	**0.028**	**0.031**
All instances		**10,073**	**10,104**	**+31**	**288,695**	**0.044**	**0.044**	**807**	**813**	**+6**	**13,706**	**0.213**	**0.214**

Table 4.6: Quality metrics by test smell. Read *OR* and *RF* as original and refactored samples, and $\Delta = RF - OR$.

		Unit size (LOC)			Assertions			Assertion density	
Test smell	**Instance**	OR	RF	Δ	OR	RF	Δ	OR	RF
Conditional test logic	28	32	20	-12	1	1	0	0.03	0.05
	29	17	25	+8	2	4	+2	0.12	0.16
	32	33	27	-6	1	3	+2	0.03	0.11
Total		**82**	**72**	**-10**	**4**	**8**	**+4**	**0.06**	**0.11**
Eager test	18	313	374	+61	231	231	0	0.74	0.62
	19	329	390	+61	239	239	0	0.73	0.61
	20	318	347	+29	188	188	0	0.59	0.54
	22	280	194	-86	43	62	+19	0.15	0.32
	31	55	82	+27	14	14	0	0.25	0.17
	35	363	405	+42	332	332	0	0.91	0.82
Total		**1,658**	**1,792**	**+134**	**1,047**	**1,066**	**+19**	**0.56**	**0.51**
Empty test	24	14	8	-6	1	1	0	0.07	0.12
Total		**14**	**8**	**-6**	**1**	**1**	**0**	**0.07**	**0.12**
Empty test (fixtures)	11	35	29	-6	5	5	0	0.14	0.17
	12	63	57	-6	7	7	0	0.11	0.12
	14	151	145	-6	24	24	0	0.16	0.17
Total		**249**	**231**	**-18**	**36**	**36**	**0**	**0.14**	**0.15**
General fixture	1	1,187	1,198	+11	324	324	0	0.27	0.27
	9	828	918	+90	172	172	0	0.21	0.19
	10	65	64	-1	18	18	0	0.28	0.28
	26	176	197	+21	40	40	0	0.23	0.20
	27	157	156	-1	26	26	0	0.17	0.17
Total		**2,413**	**2,533**	**+120**	**580**	**580**	**0**	**0.23**	**0.22**
Ignored test	4	2,672	88	-2,584	11	11	0	0.00	0.12
	5	791	79	-712	1	1	0	0.00	0.01
	6	6,707	39	-6,668	13	13	0	0.00	0.33
	7	1,041	11	-1,030	3	3	0	0.00	0.27
	8	597	45	-552	1	0	-1	0.00	0.00
Total		**11,808**	**262**	**-11,546**	**29**	**28**	**-1**	**0.00**	**0.15**
Lazy test (JUnit 4)	17	289	96	-193	81	3	-78	0.28	0.03
	21	224	88	-136	54	3	-51	0.24	0.03
	23	405	527	+122	332	18	-314	0.82	0.03
	30	98	29	-69	26	16	-10	0.27	0.55
	39	1,544	92	-1,452	1,440	12	-1,428	0.93	0.13
	40	95	85	-10	22	2	-20	0.23	0.02
Total		**2,655**	**917**	**-1,738**	**1,955**	**54**	**-1,901**	**0.46**	**0.13**
Lazy test (JUnit 5)	3	1,024	1,183	+159	502	63	-439	0.49	0.05
	33	66	24	-42	24	3	-21	0.36	0.12
	34	104	68	-36	53	1	-52	0.51	0.01
	36	101	53	-48	36	1	-35	0.36	0.02
	37	390	127	-263	157	5	-152	0.40	0.04
Total		**1,685**	**1,455**	**-230**	**772**	**73**	**-699**	**0.42**	**0.05**
Redundant print	25	150	123	-27	31	31	0	0.21	0.25
Total		**150**	**123**	**-27**	**31**	**31**	**0**	**0.21**	**0.25**
Unknown test	2	266	283	+17	0	19	+19	0.00	0.07
	13	34	36	+2	0	2	+2	0.00	0.06
	15	46	73	+27	0	27	+27	0.00	0.37
	16	24	28	+4	2	4	+2	0.08	0.14
	38	49	54	+5	0	4	+4	0.00	0.07
Total		**419**	**474**	**+55**	**2**	**56**	**+54**	**0.02**	**0.14**
All instances		**21,133**	**7,867**	**-13,266**	**4,457**	**1,933**	**-2,524**	**0.26**	**0.20**

Chapter 5

Discussion

5.1 Revisiting the research questions

This section synthesises the empirical findings by directly revisiting each of the three research questions.

5.1.1 *RQ1: To what extent does refactoring test smells affect the energy consumption of JUnit test suites?*

The empirical findings indicate that the impact of refactoring test smells on energy consumption depends on the specific smell type. Crucially, only two of the evaluated smells demonstrated a substantial percentage difference in energy consumption following refactoring: *Ignored test* and *Lazy test (JUnit 5)*.

Both of these offer distinct architectural insights. For *Ignored test*, the substantial energy savings suggest that completely eliminating ignored tests significantly reduces the systemic overhead associated with test discovery and execution. Conversely, for *Lazy test (JUnit 5)*, the results reveal that migrating to `@ParameterizedTest` was associated with a large runtime overhead in our evaluated instances. For these two test smells, the sustainability impacts are clear and direct. For the remaining test smells, however, the energy effects were markedly smaller or inconsistent, so actionable conclusions must be approached with caution. The recommendations that follow reflect this uncertainty.

Nevertheless, several marginal trends emerge that carry practical significance. For instance, while *Lazy test (JUnit 4)* instances exhibited lower energy consumption post-refactoring, alongside broader quality improvements after parameterisation, these energy gains were marginal. In this scenario, refactoring would be more justified by internal code quality and maintainability improvements rather than by savings in energy consumption. Conversely, all *Eager test* instances point to marginally higher energy consumption after refactoring, so the decision to refactor introduces a direct trade-off, hinging on whether the expected structural quality benefits outweigh the higher energy costs. For *Unknown test*, *Conditional test logic*, and *General fixture*, the results show heterogeneous and highly context-dependent differences, while for smells such as *Redundant print* and *Empty test*, the

number of available instances was too small to support meaningful or statistically generalisable conclusions.

5.1.2 *RQ2: What is the relationship between changes in energy consumption and changes in execution time following refactoring?*

Overall, absolute correlations between energy and time seem to be pronounced across most test smells, though slightly attenuated in specific cases like *Lazy test (JUnit 5)*, likely due to background processes that mask the relationship in raw numbers. However, evaluating relative percentage differences to mitigate this noise yields an overall strong correlation ($\rho(\Delta\%E, \Delta\%t) = 0.765$), becoming even stronger within (highlighted) smells that exhibit an energy difference of more than 5% ($\rho = 0.988$, $p < 0.001$). Building on this, no instances (and thus no smell types) exhibit energy significantly dropping while execution time stays the same, or vice versa.

A plausible explanation for this lies in the physical constraints of processor architecture. Because energy is the product of power and time ($E = P \times t$), a reduction in energy without a corresponding reduction in execution time would require a significant drop in the CPU's average power draw. Since our test refactorings primarily alter instruction volume rather than low-level hardware resource usage, the power profile remains relatively constant during execution.

From this, we conclude that energy efficiency in test suites is closely coupled with execution efficiency within our experimental setting, even when refactoring is primarily motivated by readability and maintainability concerns. These findings suggest that green software refactoring may not be entirely separable from performance optimisation. In our study, improvements in execution time were consistently associated with reductions in energy consumption, indicating that execution-time optimisation is an effective and accessible strategy for improving the energy efficiency of test suites.

5.1.3 *RQ3: To what extent do test smell refactorings preserve or improve functional and structural test quality?*

The evaluation of test quality revealed that our control metrics, i.e., coverage and mutation scores, remained stable across the vast majority of instances. This indicates that our refactorings successfully acted as experimental controls, largely preserving the fault-detection capabilities captured by these metrics, without causing behavioural regressions. The few instances that did exhibit changes were minor and expected, occurring as a direct consequence of adding explicit assertions or parameterising a test class. Meanwhile, the structural changes, including lines of code and assertion counts, moved predictably and aligned with the architectural design of each refactoring strategy. For example, unit size and explicit assertions shrank when removing *Ignored test* methods, but increased when *Eager test* methods were split and extracted into their own dedicated methods.

Overall, these results suggest that test smell refactorings can be applied without materially affecting the fault-detection capabilities captured by our control metrics. Although the observed structural changes were consistent with the intended design goals of each refactor-

ing strategy, they should not be interpreted as evidence of a universal improvement in test quality. Rather, the findings indicate that maintainability-oriented refactorings can be introduced without substantially compromising the functionality or bug-detection effectiveness measured in this study.

5.2 Practical implications

Based on the empirical findings addressing the research questions, this section presents actionable recommendations for practitioners, tool developers, researchers, and educators in this field. Ultimately, a key takeaway of this study is that, under our specific experimental conditions, in a controlled environment containing single machine executing JUnit test cases sequentially on a CPU, changes in execution time seem to be strongly correlated with energy consumption. Our findings confirm that, for the investigated test smells under this specific context, the corresponding refactoring strategies that successfully reduce execution time will most likely translate into energy savings too, and the inverse holds true as well.

Furthermore, measuring energy consumption to promote sustainability is itself energy-intensive, ironically. This is an inevitable cost of green software research, yet the intention is that improved understanding enables larger savings than the energy spent on measurement. To put this into perspective, a measurement session (for a single instance) in this study often took more than 30 minutes, with each run consuming about 426 J on average. As such, this study consumed a total of 1 MJ[1], excluding discarded runs and energy outside the measurement scope. This total energy consumption is equivalent to running a 10 W LED bulb for roughly 28 hours, or boiling approximately 3 L of water[2]. While this may not seem like much for a standalone experiment, these values represent a single localised instance; when multiplied across dozens of software repositories in high-frequency Continuous Integration (CI/CD) environments where tests execute thousands of times daily, these minor measurement footprints scale into significant energy overheads.

5.2.1 For practitioners

Translating the results into practical insights requires carefully weighing execution time, energy efficiency, and code quality against one another. To assist developers in navigating these trade-offs, Table 5.1 provides a targeted breakdown of the specific quality impacts, energy impacts, and actionable empirical recommendations for each evaluated test smell.

We recommend removing *Ignored tests* when they are unused or no longer relevant. While there are legitimate cases for temporary disabling or conditional execution, retaining obsolete ignored tests reduces the readability of a test class, as developers must establish whether and why a test runs. The reduction in unit size supports this view. Coincidentally, these tests also increase energy use, making removal a straightforward win for both energy and quality. Conversely, for *Lazy test (JUnit 5)*, which involves parameterising tests using the `@ParameterizedTest` annotation, we advise caution when applying this refactor-

[1] 40 instances $\times$ 60 runs $\times$ 425 J

[2] Assuming $c = 4184\,\mathrm{J\,kg^{-1}\,K^{-1}}$ and $\Delta T = 100 - 18.5 = 81.5\,^{\circ}\mathrm{C}$

Table 5.1: Summary of quality and energy impacts by smell type, together with their recommendation.

Test smell	Quality impact	Energy impact	Recommendation
Eager test	Increase in unit size, but improved maintainability.	Marginal energy increase.	**Situational:** refactor only if structural readability outweighs the (minor) energy costs.
General fixture	Minimal quantifiable quality differences, but reduced setup and teardown overhead.	Heterogeneous energy impacts.	**Caution:** refactor based on system guidelines; energy impacts are not conclusive.
Ignored test	Removal of obsolete methods and reduction in unit size.	Energy savings (roughly 16%).	**Recommended:** remove obsolete and unused ignored or disabled tests.
Lazy test (JUnit 4)	Structural quality improvements through parameterisation and reduced unit size (LOC).	Marginal energy savings.	**Situational:** justified primarily for internal code quality, not energy savings.
Lazy test (JUnit 5)	Structural quality improvements through parameterisation and reduced unit size (LOC).	Energy increase (roughly 20%).	**Discouraged:** avoid parameterised refactorings unless there are large maintainability gains to be had.
Other smells	Context-dependent.	Negligible or inconclusive impacts.	**Caution:** refactor based on system guidelines; energy impacts are not conclusive.

ing solely for maintainability reasons. Although it may improve internal code quality, our results indicate that it can substantially increase both execution time and energy consumption. Consequently, the maintainability benefits should be weighed against these potential efficiency costs.

Practically, developers may balance maintainability and energy footprints based on their deployment environment. In high-frequency CI/CD pipelines, minor energy penalties compound into significant environmental and infrastructure costs. In these environments, execution efficiency should be prioritised over design patterns. Conversely, for infrequently executed test suites where human comprehension is the primary bottleneck, the readability gains of thorough refactoring heavily outweigh periodic energy overheads, making quality-centric code refactorings the better approach.

5.2.2 For tool developers

Automated test smell detection, and perhaps refactoring, should be integrated directly into the developer workflow in the IDE (e.g., SonarQube, IntelliJ) to provide real-time warnings. Crucially, tool developers could not only flag a smell based on code maintainability, but also provide energy-aware impact warnings. For instance, if a developer triggers a `@ParameterizedTest` refactoring, the IDE could display a subtle warning noting the potential overhead in execution time and energy footprint, allowing the practitioner to make an informed trade-off.

Moreover, the manual effort required for refactoring suggests a strong need for automated support. This could be provided through rule-based approaches, integration into existing detection tools, or the use of large language models (LLMs). While artificial intelligence (AI) introduces its own structural limitations and environmental footprints, its deployment in this study for generating suggestions and handling repetitive changes indicates that it could play a role in (energy-aware) refactoring of test suites.

5.2.3 For researchers

As mentioned previously, our findings demonstrate that execution time functions as an effective proxy for energy consumption under our specific conditions. Researchers operating under similar constraints may want to bypass specialised, time-intensive energy profiling tools such as *EnergiBridge*, because within this narrow scope, namely, test cases running sequentially on a single machine, execution time may provide sufficient empirical guidance.

5.2.4 For educators

Software engineering curricula provide an ideal foundation for introducing green software engineering concepts early on, teaching students that code and sustainability, or energy footprints specifically, are intrinsically linked. Cultivating this mindset early not only prevents developers from introducing (energetic) anti-patterns into (industrial) software systems, but also drives widespread awareness of sustainable software development practices.

5.3 Threats to validity

The goal of our study was to understand test smell refactorings and measure their energy behaviour, providing insight into how specific refactorings affect specific test smells and how these effects compare, mainly in terms of energy consumption and internal quality. In this section, we present some threats to the validity of our study, divided into four categories, namely: construct validity, internal validity, conclusion validity, and external validity.

5.3.1 Construct validity

A significant threat to validity centres on human subjectivity in the experimental design. First, the manual nature of the refactorings introduces a threat to behavioural equivalence, as a researcher's subjective interpretation could inadvertently alter a test suite's original logic. We mitigated this by performing strict manual validation alongside code coverage and mutation testing to quantitatively verify that functional behaviour remained unchanged. Second, distinct implementation choices can yield varying energy profiles despite identical test behaviour. Concretely, because energy consumption is highly sensitive to coding styles and implementation choices, another developer might achieve the same functional results with a different energy footprint. We minimised this inherent variability by strictly adhering to standard refactoring patterns sourced from literature. Similarly, our use of internal quality

through unit size and assertion density relies on imperfect proxies that may not fully reflect human perception of tests, which is why manual assessment was used as well.

Additionally, a threat arises from the scope of our experimental unit. Our experiments focus strictly on isolated smell instances rather than full, system-level test suites. By symmetrically removing all unrelated test methods to isolate the targeted smell, our measurements capture the localised effects of refactorings. While this isolation was necessary to eliminate confounding factors and establish clear internal controls, it introduces a construct threat regarding scalability; in a real-world deployment, these local energy variations might be diluted or masked by the broader execution overhead of a full-suite execution.

Finally, construct validity may be impacted by our measurement and detection tooling stack. Energy consumption was measured via EnergiBridge using hardware counters and sampled telemetry. While this provides a practical, repeatable proxy for energy use, it remains an indirect measurement sensitive to sampling granularity and external system components (e.g., background services or peripheral activity) not fully captured by the counters. To mitigate this, we executed a high volume of experimental repetitions and subjected the resulting dataset to rigorous statistical testing. In addition, our initial automated smell detection relies on *tsDetect*, which, although it reports high accuracy, can still yield false positives or miss context-specific smells, and this consequently affects which instances were selected and refactored. Our manual validation phase reduced this risk by retaining only confirmed instances, though the possibility of false negatives remains.

5.3.2 Internal validity

Another issue is that grouping smells by type can mask instance-specific characteristics. Individual tests may have unique execution profiles that are lost when aggregated by smell, which can obscure differences in energy behaviour. We mitigated this by defining each smell type narrowly enough to apply a consistent refactoring strategy, and by splitting smell variants (e.g., *Lazy test* and *Empty test*), though this sometimes reduced the number of instances per subtype.

Although we established a rigorous experimental setup to control for external variables, energy measurements remain fundamentally sensitive to background operating system activity, thermal throttling, and JVM runtime fluctuations. Our pipeline actively mitigates these factors through structured warm-up, randomisation, and cooldown phases. However, because complete isolation is impossible, some degree of residual noise is inevitable. This limitation is further underscored by the use of Windows, which carries a non-trivial baseline resource footprint, and a different operating system or machine may have provided a cleaner isolation of energy consumption. Nonetheless, even Linux and similar systems have background daemons and services that cannot always be fully disabled, so baseline activity remains a concern across platforms.

5.3.3 Conclusion validity

A primary threat to conclusion validity stems from the limited sample sizes available for test smells. For several smell types, only a small number of refactorable instances could

be found within the target systems, which naturally restricts the statistical power of the corresponding tests. We sought to temper this limitation by executing 30 independent experimental runs for both the original and refactored versions of each instance, increasing the volume of data. Crucially, the assumptions underpinning these tests were well-supported by our dataset: all evaluated instances satisfied normality, and outlier rates remained consistently low across our experiments.

5.3.4 External validity

To support stronger generalisations, a larger set of instances across a broader range of systems would be required. The characteristics of our available systems limited the achievable sample size for certain test smells. Moreover, the evaluated test suites came from a specific set of small, open-source Java systems, and thus our findings may not transfer to large industrial systems, other programming languages, or different domains.

5.4 Generative AI disclosure

During the preparation of this work, the author made use of generative AI tools, including *Gemini 3.5 Flash* for brainstorming, idea generation, grammar and spelling checks, and paraphrasing and rewording. Additionally, *GPT-5.3-Codex* was employed for code assistance, explanations, and debugging. All AI-generated outputs, including text, code, and analytical results, were reviewed, validated, and modified by the author to ensure scientific rigour, integrity, and accuracy. The author takes full responsibility for the publication's content.

5.5 Ethical considerations

As software engineering intersects with global sustainability, researchers must critically evaluate the socio-environmental dimensions of their methodology and findings. This section outlines the ethical considerations surrounding this study.

5.5.1 Socio-technical trade-offs

Optimising software for sustainability introduces a fundamental tension between machine efficiency, human cognitive limits, and system reliability. Our findings show that resolving the *Lazy test* smell using the `@ParameterizedTest` annotation increases energy consumption despite improving code maintainability. Imposing strict time and energy-minimisation guidelines risks shifting these burdens onto developers through complex, hard-to-maintain architectures, increasing cognitive load. Furthermore, green refactoring must never compromise software safety. To ensure that reducing runtime footprints did not silently introduce regressions or degrade bug-catching capabilities, this study implemented code coverage and mutation testing as strict experimental controls, verifying that functional dependability remained largely stable post-refactoring.

5.5.2 Proxy reliability

While this study establishes execution time as a reliable proxy for energy consumption, treating it as a flawless equivalent presents distinct ethical hazards. Practitioners may oversimplify this correlation by assuming all fast code is green code, masking dynamic hardware power spikes, or worse, justifying dangerous reductions in test coverage under the guise of 'saving energy'. These findings must therefore be framed transparently as micro-optimisations that complement, rather than substitute for, comprehensive green systems design.

5.5.3 Environmental justice

According to Aczel et al. [2] and their report on the environmental costs of digital infrastructure, the global expansion of computing relies on a severe, threefold resource extraction cycle characterised by expanding carbon emissions, massive regional water consumption for thermal cooling, and intense localised land degradation. Ironically, empirical sustainability research suffers from this too, in that we expend localised energy to prove or enhance long-term software efficiency. In this study specifically, we tried to mitigate CPU and DRAM footprints by capping target system sizes at 15 MB to eliminate compile-time waste and invoking JUnit directly to avoid the overhead of Maven test goals. More broadly, it is important to recognise that monitoring and measuring energy consumption itself inevitably requires resources and therefore consumes a small amount of energy; this trade-off is not unique to sustainability research but applies to energy-tracking and optimisation efforts in general. Minimising the energy consumption of software is an ethical duty tied directly to environmental justice, as the physical burdens of cloud computing are disproportionately absorbed by localised communities hosting data centres, while the economic benefits are reaped elsewhere [2].

Chapter 6

Conclusion

This study investigated the intersection of green software engineering and software quality by evaluating how refactoring test smells affects energy consumption, execution time, and test quality in Java JUnit test suites. By pairing automated smell detection with systematic, literature-backed manual refactorings, this work provides empirical clarity on the sustainability impacts of refactored test smells.

(RQ1) The findings reveal that the energy impacts of refactored tests are highly smell-specific, with median energy changes concentrated in two test smells: *Ignored test*, with a decrease of roughly 16%, and *Lazy test (JUnit 5)*, with an increase of roughly 20%. Although several effects were statistically significant, the absolute energy differences were often modest, meaning their practical relevance depends heavily on the execution context. In high-frequency CI/CD pipelines, where test suites run hundreds of times per day, even small overheads can accumulate into substantial infrastructure costs and carbon emissions. For example, the 20% energy increase associated with migration to the JUnit 5 `@ParameterizedTest` annotation may become consequential at scale. Conversely, for infrequently executed test suites, where human comprehension is the primary concern, the benefits of refactoring are likely to outweigh periodic energy overheads.

(RQ2) The results also show that energy effects are closely tied to execution efficiency within our evaluation context, likely due to a relatively stable hardware power profile across our experimental runs. This suggests that execution time may serve as a practical proxy for energy consumption in similar settings. **(RQ3)** Finally, our control metrics indicate that the structural refactorings used in this study can be applied without degrading test functionality.

As a result, this study makes four main contributions. First, it provides an empirical investigation of how test smell refactoring affects energy consumption in Java JUnit test suites. Second, it curates a dataset of 40 manually validated and refactored test smell instances. Third, it presents a reproducible measurement pipeline for evaluating energy, execution time, and test-quality controls at the smell-instance level, through our replication package [12]. Fourth, it translates these findings into smell-specific, actionable guidance (Table 5.1) that helps practitioners and tool developers weigh energy versus quality when deciding whether and how to refactor test smells. Ultimately, integrating these insights into automated refactoring tools could provide practitioners with energy-aware feedback, embedding sustainability directly into the (continuous) software lifecycle.

6.1 Future work

Several promising avenues for future research emerge from the limitations and findings of this study. First, a primary constraint of current test smell research is its heavy reliance on the Java ecosystem, driven by the maturity of tooling built around it. This tooling gap is increasingly problematic given the widespread industry adoption of dynamically typed languages such as Python and JavaScript [3, 80]. Future work could focus on developing detection tools for these dynamic ecosystems, where test architecture and execution paradigms fundamentally differ from typed, object-oriented environments.

Second, to increase the generalisability of our results, future iterations could incorporate larger, more balanced test smell distributions across a wider set of (open-source) systems, alleviating potential system-specific biases in the dataset. Similarly, future work should transition to full-suite deployments, as this study evaluated isolated smell instances to capture local refactorings. Measuring entire, production-grade test suites will clarify whether the observed energy savings scale proportionally or become diluted by broader execution overhead.

Finally, future studies should aim to replicate and validate our findings across more diverse system architectures. Because this study deliberately constrained its evaluation to a single machine executing test cases sequentially on a CPU, an open question remains regarding how tightly coupled execution time and energy consumption remain in complex environments. Future iterations of this work could investigate these relationships within parallel, multi-threaded applications, *Continuous Integration* (CI) pipelines, or containerised architectures (e.g., *Docker* or *Kubernetes*).

Bibliography

[1] Marwen Abbes, Foutse Khomh, Yann-Gaël Guéhéneuc, and Giuliano Antoniol. An empirical study of the impact of two antipatterns, blob and spaghetti code, on program comprehension. In *Proceedings of the 15th European Conference on Software Maintenance and Reengineering (CSMR)*, pages 181–190. IEEE, 2011.

[2] M. Aczel, S. Chamanara, M. Matin, A. Farsi, T. Marwala, and K. Madani. Environmental cost of AI's energy use: Carbon, water and land footprints. Unu-inweh report, United Nations University Institute for Water, Environment and Health (UNU-INWEH), Richmond Hill, Ontario, Canada, 2026.

[3] Wajdi Aljedaani, Anthony Peruma, Ahmed Aljohani, Mazen Alotaibi, Mohamed Wiem Mkaouer, Ali Ouni, Christian D. Newman, Abdullatif Ghallab, and Stephanie Ludi. Test smell detection tools: a systematic mapping study. In *Proceedings of the 25th International Conference on Evaluation and Assessment in Software Engineering (EASE)*, pages 170–180, 2021.

[4] Eman Abdullah AlOmar, Hussein Alrubaye, Mohamed Wiem Mkaouer, Ali Ouni, and Marouane Kessentini. Refactoring practices in the context of modern code review: An industrial case study at xerox. In *2021 IEEE/ACM 43rd International Conference on Software Engineering: Software Engineering in Practice (ICSE-SEIP)*, pages 348–357. IEEE, 2021. doi: 10.1109/ICSE-SEIP52600.2021.00044.

[5] Maurício Aniche. *Effective Software Testing: A developer's guide*. Simon and Schuster, 2022.

[6] Robert Arntzenius, Xutong Liu, and Andy Zaidman. On the energy consumption of continuous integration in open-source java projects. In *2026 IEEE International Conference on Software Analysis, Evolution and Reengineering - Companion (SANER-C)*, pages 181–188, Los Alamitos, CA, USA, 2026. IEEE Computer Society. doi: 10.1109/SANER-C67878.2026.00030.

[7] Paul Baker, Dominic Evans, Jens Grabowski, Helmut Neukirchen, and Benjamin Zeiss. Trex - the refactoring and metrics tool for ttcn-3 test specifications. In *Test-*

ing: Academic & Industrial Conference-Practice And Research Techniques (TAIC PART'06), pages 90–94. IEEE, 2006. doi: 10.1109/TAIC-PART.2006.11.

[8] Gabriele Bavota, Abdallah Qusef, Rocco Oliveto, Andrea De Lucia, and David Binkley. An empirical analysis of the distribution of unit test smells and their impact on software maintenance. In *Proceedings of the 2012 28th IEEE International Conference on Software Maintenance (ICSM)*, pages 56–65. IEEE, 2012.

[9] Gabriele Bavota, Andrea De Lucia, Massimiliano Di Penta, Rocco Oliveto, and Fabio Palomba. An experimental investigation on the innate relationship between quality and refactoring. *Journal of Systems and Software*, 107:1–14, 2015. doi: 10.1016/j.jss.2015.05.017.

[10] Gabriele Bavota, Abdallah Qusef, Rocco Oliveto, Andrea De Lucia, and David Binkley. Are test smells really harmful? an empirical study. *Empirical Software Engineering*, 20(4):1052–1094, 2015.

[11] Kent Beck. Code smell. WikiWikiWeb, 2014. URL `http://wiki.c2.com/?CodeSmell`.

[12] Simon Biennier. Replication package for "is cleaner greener? on the energy impact of refactored test smells", 2026. URL `https://doi.org/10.5281/zenodo.20721696`.

[13] Bogdan Bodnar. La promesse d'une intelligence artificielle « verte » se heurte à la boulimie des usages. *La Tribune*, mar 2026. URL `https://www.latribune.fr/article/tech/17696212098690/la-promesse-dune-intelligence-artificielle-verte-se-heurte-a-la-boulimie-des-usages`.

[14] Manuel Breugelmans and Bart van Rompaey. TestQ: Exploring structural and maintenance characteristics of unit test suites. In *Proceedings of the 1st International Workshop on Advanced Software Development Tools and Techniques (WASDeTT-1)*, Antwerp, Belgium, 2008.

[15] Coral Calero and Mario Piattini. Puzzling out software sustainability. *Sustainable Computing: Informatics and Systems*, 16:117–124, 2017. doi: 10.1016/j.suscom.2017.10.011.

[16] Zhongyan Chen, Suzanne M. Embury, and Markel Vigo. Who is afraid of Test Smells? Assessing technical debt from developer actions. In *Testing Software and Systems: 35th IFIP WG 6.1 International Conference, ICTSS 2023*, volume 14131 of *Lecture Notes in Computer Science*, pages 171–189. Springer, 2023. doi: 10.1007/978-3-031-43240-8_11.

[17] Andrew A. Chien. Owning computing's environmental impact. *Communications of the ACM*, 62(3):5–5, 2019. doi: 10.1145/3310359.

[18] Jacob Cohen. *Statistical power analysis for the behavioral sciences*. Lawrence Erlbaum Associates, 2 edition, 1988. ISBN 9780805802832.

[19] Henry Coles, Thomas Laurent, Christopher Henard, Mike Papadakis, and Anthony Ventresque. Pit: a practical mutation testing tool for java (demo). In *Proceedings of the 25th International Symposium on Software Testing and Analysis*, ISSTA 2016, pages 449–452, New York, NY, USA, 2016. Association for Computing Machinery. ISBN 9781450343909. doi: 10.1145/2931037.2948707. URL https://doi.org/10.1145/2931037.2948707.

[20] Luís Cruz. Green software engineering done right: a scientific guide to set up energy efficiency experiments, 2021. URL http://luiscruz.github.io/2021/10/10/scientific-guide.html. Blog post.

[21] Luís Cruz. Tools to measure software energy consumption from your computer, 2021. URL http://luiscruz.github.io/2021/07/20/measuring-energy.html. Blog post.

[22] Luís Cruz and Rui Abreu. On the energy footprint of mobile testing frameworks. *IEEE Transactions on Software Engineering*, 47(10):2260–2271, 2021. doi: 10.1109/TSE.2019.2946163.

[23] Luís Cruz and Petra Heck. Lessons learned from developing green software. In *Moral Design and Green Technology*, pages 168–185. Brill, 2025. doi: 10.3920/9789004730779_012.

[24] Dario Di Nucci, Fabio Palomba, Antonio Prota, Annibale Panichella, Andy Zaidman, and Andrea De Lucia. Petra: a software-based tool for estimating the energy profile of android applications. In *Proceedings of the 39th International Conference on Software Engineering Companion*, ICSE-C '17, pages 3–6. IEEE Press, 2017. ISBN 9781538615898. doi: 10.1109/ICSE-C.2017.18. URL https://doi.org/10.1109/ICSE-C.2017.18.

[25] Pierre Dragicevic. A mean difference is an effect size. Research Report RR-9354, Inria Saclay Ile de France, 2020.

[26] Ericsson. Sustainability and ICT. Ericsson Mobility Report, 2023. URL https://www.ericsson.com/en/reports-and-papers/mobility-report/dataforecasts/ict-sustainability.

[27] Martin Fowler. *Refactoring: Improving the Design of Existing Code*. Addison-Wesley, 1999.

[28] Martin Fowler. Code smell, February 2006. URL https://martinfowler.com/bliki/CodeSmell.html.

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

[30] Michaela Greiler, Andy Zaidman, Arie van Deursen, and Margaret-Anne Storey. Strategies for avoiding test fixture smells during software evolution. In *2013 10th Working Conference on Mining Software Repositories (MSR)*, pages 387–396. IEEE, 2013. doi: 10.1109/MSR.2013.6624053.

[31] Marco Gribaudo, Thi Thao Nguyen Ho, Barbara Pernici, and Giuseppe Serazzi. Analysis of the influence of application deployment on energy consumption. In *International Workshop on Energy Efficient Data Centers*, 2014.

[32] Foyzul Hassan, Shaikh Mostafa, Edmund S. L. Lam, and Xiaoyin Wang. Automatic building of Java projects in software repositories: A study on feasibility and challenges. In *2017 ACM/IEEE International Symposium on Empirical Software Engineering and Measurement (ESEM)*, pages 38–47. IEEE, 2017. doi: 10.1109/ESEM.2017.10.

[33] Abram Hindle. Green mining: a methodology of relating software change and configuration to power consumption. *Empirical Softw. Engg.*, 20(2):374–409, April 2015. ISSN 1382-3256. doi: 10.1007/s10664-013-9276-6. URL https://doi.org/10.1007/s10664-013-9276-6.

[34] Boris Iglewicz and David C. Hoaglin. *How to Detect and Handle Outliers*, volume 16 of *ASQC basic references in quality control: statistical techniques*. ASQC Quality Press, 1993.

[35] René Just, Darioush Jalali, and Michael D. Ernst. Defects4j: a database of existing faults to enable controlled testing studies for java programs. In *Proceedings of the 2014 International Symposium on Software Testing and Analysis*, ISSTA 2014, pages 437–440, New York, NY, USA, 2014. Association for Computing Machinery. ISBN 9781450326452. doi: 10.1145/2610384.2628055. URL https://doi.org/10.1145/2610384.2628055.

[36] Ken Kelley and Kristopher J. Preacher. On effect size. *Psychological Methods*, 17(2):137–152, jun 2012. doi: 10.1037/a0028086.

[37] Kashif Nizam Khan, Mikael Hirki, Tapio Niemi, Jukka K. Nurminen, and Zhonghong Ou. Rapl in action: Experiences in using rapl for power measurements. *ACM Trans. Model. Perform. Eval. Comput. Syst.*, 3(2), March 2018. ISSN 2376-3639. doi: 10.1145/3177754. URL https://doi.org/10.1145/3177754.

[38] Ali Khatami and Andy Zaidman. State-of-the-practice in quality assurance in java-based open source software development. *Software: Practice and Experience*, 54(8):1408–1446, 2024. doi: 10.1002/spe.3321.

[39] Foutse Khomh, Massimiliano Di Penta, Yann-Gaël Guéhéneuc, and Giuliano Antoniol. An exploratory study of the impact of antipatterns on class change- and fault-proneness. *Empirical Software Engineering*, 17(3):243–275, 2012.

[40] Dong Jae Kim. An empirical study on the evolution of test smell. In *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering: Companion Proceedings*, ICSE '20, pages 149–151, New York, NY, USA, 2020. Association for Computing Machinery. ISBN 9781450371223. doi: 10.1145/3377812.3382176.

[41] Dong Jae Kim, Tse-Hsun (Peter) Chen, and Jinqiu Yang. The secret life of test smells – an empirical study on test smell evolution and maintenance. *Empirical Software Engineering*, 26(5):100, 2021. doi: 10.1007/s10664-021-09969-1.

[42] Miryung Kim, Thomas Zimmermann, and Nachiappan Nagappan. A field study of refactoring challenges and benefits. In *Proceedings of the 20th ACM SIGSOFT Symposium on the Foundations of Software Engineering (FSE)*, page 50. ACM, 2012. doi: 10.1145/2393596.2393655.

[43] Stefano Lambiase, Andrea Cupito, Fabiano Pecorelli, Andrea De Lucia, and Fabio Palomba. Just-in-time test smell detection and refactoring: The DARTS project. In *Proceedings of the 28th International Conference on Program Comprehension (ICPC)*, pages 441–445, 2020. doi: 10.1145/3387904.3389248.

[44] Gustavo Lopes, Davi Romão, Elvys Soares, Márcio Ribeiro, Guilherme Amaral, Rohit Gheyi, and Ivan Machado. A road to find them all: Towards an agnostic strategy for test smell detection. In *Proceedings of the XXIII Brazilian Symposium on Software Quality (SBQS '24)*, pages 231–241, New York, NY, USA, 2024. Association for Computing Machinery. doi: 10.1145/3701625.3701662.

[45] Mika V. Mäntylä and Casper Lassenius. Subjective evaluation of software evolvability using code smells: An empirical study. *Empirical Software Engineering*, 11(3):395–431, 2006. doi: 10.1007/s10664-006-9002-8.

[46] Cosmin Marsavina, Daniele Romano, and Andy Zaidman. Studying fine-grained co-evolution patterns of production and test code. In *2014 IEEE 14th International Working Conference on Source Code Analysis and Manipulation*, pages 195–204. IEEE, 2014.

[47] Luana Martins, Taher A. Ghaleb, Heitor Costa, and Ivan Machado. A comprehensive catalog of refactoring strategies to handle test smells in java-based systems. *Software Quality Journal*, 32(3):641–679, 2024. doi: 10.1007/s11219-024-09663-7.

[48] Gerard Meszaros. *xUnit Test Patterns: Refactoring Test Code*. Addison-Wesley Professional, Boston, MA, USA, 2007.

[49] Raimund Moser, Alberto Sillitti, Pekka Abrahamsson, and Giancarlo Succi. Does refactoring improve reusability? In *International Conference on Software Reuse*, volume 4039, pages 287–297. Springer, 2006. doi: 10.1007/11766155_22.

[50] Mountainminds GmbH & Co. KG and Contributors. *JaCoCo Java Code Coverage Library*, 2026. URL `https://www.jacoco.org/jacoco/`.

[51] Nicholas Alexandre Nagy and Rabe Abdalkareem. On the co-occurrence of refactoring of test and source code. In *2022 IEEE/ACM 19th International Conference on Mining Software Repositories (MSR)*, pages 122–126. IEEE, 2022. doi: 10.1145/3524842.3528477.

[52] Annibale Panichella, Sebastiano Panichella, Gordon Fraser, Anand Ashok Sawant, and Vincent J. Hellendoorn. Revisiting test smells in automatically generated tests: Limitations, pitfalls, and opportunities. In *2020 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, pages 523–533. IEEE, 2020. doi: 10.1109/ICSME46990.2020.00056.

[53] Annibale Panichella, Sebastiano Panichella, Gordon Fraser, Anand Ashok Sawant, and Vincent J. Hellendoorn. Test smells 20 years later: detectability, validity, and reliability. *Empirical Software Engineering*, 27(7):170, 2022. doi: 10.1007/s10664-022-10207-5.

[54] Jevgenija Pantiuchina, Fiorella Zampetti, Simone Scalabrino, Valentina Piantadosi, Rocco Oliveto, Gabriele Bavota, and Massimiliano Di Penta. Why developers refactor source code: A mining-based study. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 29(4):1–30, 2020. doi: 10.1145/3402450.

[55] Abhinav Pathak, Y. Charlie Hu, and Ming Zhang. Where is the energy spent inside my app? fine grained energy accounting on smartphones with eprof. In *Proceedings of the 7th ACM European Conference on Computer Systems*, EuroSys '12, pages 29–42, New York, NY, USA, 2012. Association for Computing Machinery. ISBN 9781450312233. doi: 10.1145/2168836.2168841. URL https://doi.org/10.1145/2168836.2168841.

[56] Priyavanshi Pathania, Nikhil Bamby, Rohit Mehra, Samarth Sikand, Vibhu Saujanya Sharma, Vikrant Kaulgud, Sanjay Podder, and Adam P. Burden. Calculating software's energy use and carbon emissions: A survey of the state of art, challenges, and the way ahead. In *2025 IEEE/ACM 9th International Workshop on Green and Sustainable Software (GREENS)*, pages 92–99. IEEE, April 2025. doi: 10.1109/greens66463.2025.00018.

[57] Birgit Penzenstadler, Ankita Raturi, Debra Richardson, Coral Calero, Henning Femmer, and Xavier Franch. Systematic mapping study on software engineering for sustainability (se4s). In *Proceedings of the 18th International Conference on Evaluation and Assessment in Software Engineering*, pages 1–14. ACM, 2014. doi: 10.1145/2601248.2601256.

[58] Rui Pereira, Marco Couto, Francisco Ribeiro, Rui Rua, Jácome Cunha, João Paulo Fernandes, and João Saraiva. Energy efficiency across programming languages: how do energy, time, and memory relate? In *Proceedings of the 10th ACM SIGPLAN International Conference on Software Language Engineering*, SLE 2017, pages 256–267, New York, NY, USA, 2017. Association for Computing Machinery. ISBN 9781450355254. doi: 10.1145/3136014.3136031.

[59] Rui Pereira, Marco Couto, Francisco Ribeiro, Rui Rua, Jácome Cunha, João Paulo Fernandes, and João Saraiva. Ranking programming languages by energy efficiency. *Science of Computer Programming*, 205:102609, 2021. ISSN 0167-6423. doi: https://doi.org/10.1016/j.scico.2021.102609.

[60] Anthony Peruma, Khalid Almalki, Christian D. Newman, Mohamed Wiem Mkaouer, Ali Ouni, and Fabio Palomba. tsdetect: An open source test smells detection tool. In *Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, ESEC/FSE 2020, pages 1650–1654, New York, NY, USA, 2020. Association for Computing Machinery. ISBN 9781450370431. doi: 10.1145/3368089.3417921.

[61] Anthony Peruma, Steven Simmons, Eman Abdullah AlOmar, Christian D. Newman, Mohamed Wiem Mkaouer, and Ali Ouni. How do i refactor this? an empirical study on refactoring trends and topics in stack overflow. *Empirical Software Engineering*, 27(1):11, 2022. doi: 10.1007/s10664-021-10031-w.

[62] Ralph Peters and Andy Zaidman. Evaluating the lifespan of code smells using software repository mining. In *2012 16th European Conference on Software Maintenance and Reengineering (CSMR)*, pages 411–416. IEEE, 2012. doi: 10.1109/CSMR.2012.79.

[63] Gustavo Pinto and Fernando Castor. Energy efficiency: A new concern for application software developers. *Communications of the ACM*, 60(12):68–75, 2017. doi: 10.1145/3154367.

[64] Gustavo Pinto, Fernando Castor, and Yu David Liu. Understanding energy behaviors of thread management constructs. In *Proceedings of the 2014 ACM International Conference on Object Oriented Programming Systems Languages & Applications (OOPSLA)*, pages 345–360. ACM, 2014. doi: 10.1145/2660193.2660235.

[65] Valeria Pontillo, Danilo Amoroso d'Aragona, Fabiano Pecorelli, Dario Di Nucci, Filomena Ferrucci, and Fabio Palomba. Machine learning-based test smell detection. *Empirical Software Engineering*, 29(3):55, 2024. doi: 10.1007/s10664-023-10436-2.

[66] Stefan Reichhart, Tudor Gîrba, and Stéphane Ducasse. Rule-based assessment of test quality. *Journal of Object Technology*, 6(9):231–251, 2007.

[67] Gema Rodríguez-Pérez, Gregorio Robles, Alexander Serebrenik, Andy Zaidman, Daniel M. Germán, and Jesus M. Gonzalez-Barahona. How bugs are born: a model to identify how bugs are introduced in software components. *Empirical Software Engineering*, 25(2):1294–1340, 2020.

[68] Graeme D. Ruxton. The unequal variance t-test is an underused alternative to student's t-test and the mann–whitney u test. *Behavioral Ecology*, 17(4):688–690, 2006. doi: 10.1093/beheco/ark016.

[69] June Sallou, Luís Cruz, and Thomas Durieux. Energibridge: Empowering software sustainability through cross-platform energy measurement, 2023.

[70] Railana Santana, Luana Martins, Larissa Rocha, Tássio Virgínio, Adriana Cruz, Heitor Costa, and Ivan Machado. RAIDE: A tool for assertion roulette and duplicate assert identification and refactoring. In *Proceedings of the 34th Brazilian Symposium on Software Engineering (SBES '20)*, pages 374–379, New York, NY, USA, 2020. Association for Computing Machinery. doi: 10.1145/3415166.3415194.

[71] Railana Santana, Luana Martins, Larissa Rocha, Carla Illane Bezerra, Heitor Costa, and Ivan Machado. Discovering patterns in test code refactorings: A preliminary study. In *Software Engineering and Advanced Applications (SEAA 2025)*, volume 16082 of *Lecture Notes in Computer Science*, Cham, 2026. Springer. doi: 10.1007/978-3-032-04200-2_13.

[72] Patrick Schober, Christa Boer, and Lothar A. Schwarte. Correlation coefficients: Appropriate use and interpretation. *Anesthesia & Analgesia*, 126(5):1763–1768, 2018. doi: 10.1213/ANE.0000000000002864.

[73] Guilia Sellitto, Emanuele Iannone, Zadia Codabux, Valentina Lenarduzzi, Andrea De Lucia, Fabio Palomba, and Filomena Ferrucci. Toward understanding the impact of refactoring on program comprehension. In *2022 IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER)*, pages 731–742. IEEE, 2022. doi: 10.1109/SANER53432.2022.00091.

[74] S. S. Shapiro and M. B. Wilk. An analysis of variance test for normality (complete samples). *Biometrika*, 52(3/4):591–611, 1965. doi: 10.2307/2333709.

[75] Publio Silva, Carla Bezerra, Ivan Machado, and Márcio Ribeiro. AromaDr: A language-independent tool for detecting test smells. In *Anais do XXXIX Simpósio Brasileiro de Engenharia de Software (SBES 2025)*, pages 914–920, Porto Alegre, 2025. SBC. doi: 10.5753/sbes.2025.11114.

[76] Elvys Soares, Márcio Ribeiro, Rohit Gheyi, Guilherme Amaral, and André Santos. Refactoring test smells with junit 5: Why should developers keep up-to-date? *IEEE Transactions on Software Engineering*, 49(3):1152–1170, 2023. doi: 10.1109/TSE.2022.3172654.

[77] Davide Spadini, Fabio Palomba, Andy Zaidman, Magiel Bruntink, and Alberto Bacchelli. On the relation of test smells to software code quality. In *Proceedings of the 2018 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, pages 1–12. IEEE, 2018.

[78] Davide Spadini, Martin Schvarcbacher, Ana-Maria Oprescu, Magiel Bruntink, and Alberto Bacchelli. Investigating severity thresholds for test smells. In *Proceedings of the 17th International Conference on Mining Software Repositories (MSR)*, pages 311–321, 2020.

[79] C. Spearman. The proof and measurement of association between two things. *The American Journal of Psychology*, 15(1):72–101, 1904. doi: 10.2307/1412159.

[80] Stack Overflow. 2025 developer survey, 2025. URL https://survey.stackoverflow.co/2025/technology.

[81] Matúš Sulír, Michaela Bacíková, Matej Madeja, Sergej Chodarev, and Ján Juhár. Large-scale dataset of local Java software build results. *Data*, 5(3):86, 2020. doi: 10.3390/data5030086.

[82] Anne E. Trefethen and Jeyarajan Thiyagalingam. Energy-aware software: Challenges, opportunities and strategies. *Journal of Computational Science*, 4(6):444–449, 2013. doi: 10.1016/j.jocs.2013.01.005.

[83] Michele Tufano, Fabio Palomba, Gabriele Bavota, Massimiliano Di Penta, Rocco Oliveto, Andrea De Lucia, and Denys Poshyvanyk. An empirical investigation into the nature of test smells. In *Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 4–15. IEEE, 2016.

[84] Michele Tufano, Fabio Palomba, Gabriele Bavota, Rocco Oliveto, Massimiliano Di Penta, Andrea De Lucia, and Denys Poshyvanyk. When and why your code starts to smell bad (and whether the smells go away). *IEEE Transactions on Software Engineering (TSE)*, 43(11):1063–1088, 2017.

[85] Arie van Deursen, Leon Moonen, Alex van den Bergh, and Gerard Kok. Refactoring test code. In *Proceedings of the 2nd International Conference on Extreme Programming and Flexible Processes in Software Engineering (XP2001)*, pages 92–95, 2001.

[86] Roberto Verdecchia, Patricia Lago, and Carol de Vries. The future of sustainable digital infrastructures: A landscape of solutions, adoption factors, impediments, open problems, and scenarios. *Sustainable Computing: Informatics and Systems*, 35: 100767, 2022. ISSN 2210-5379. doi: https://doi.org/10.1016/j.suscom.2022.100767. URL https://www.sciencedirect.com/science/article/pii/S2210537922000889.

[87] Roberto Verdecchia Verdecchia, Emilio Cruciani, Antonia Bertolino, and Breno Miranda. Energy-aware software testing. *2025 IEEE/ACM 47th International Conference on Software Engineering: New Ideas and Emerging Results (ICSE-NIER)*, pages 101–105, 2025.

[88] Tássio Virgínio, Railana Santana, Luana Almeida Martins, Larissa Rocha Soares, Heitor Costa, and Ivan Machado. On the influence of test smells on test coverage. In *Proceedings of the XXXIII Brazilian Symposium on Software Engineering*, pages 467–471, 2019. doi: 10.1145/3350768.3352455.

[89] Vince Weaver. Linux support for power measurement interfaces. https://web.eece.maine.edu/~vweaver/projects/rapl/rapl_support.html, 2020.

[90] B. L. Welch. The generalization of student's problem when several different population variances are involved. *Biometrika*, 34(1-2):28–35, 1947. doi: 10.1093/biomet/34.1-2.28.

[91] Tomofumi Yuki and Sanjay Rajopadhye. Folklore confirmed: Compiling for speed= compiling for energy. In *Languages and Compilers for Parallel Computing*, pages 169–184. Springer, 2014. doi: 10.1007/978-3-319-09964-4_12.

[92] Andy Zaidman. An inconvenient truth in software engineering? The environmental impact of testing open source Java projects. In *Proceedings of the 5th ACM/IEEE International Conference on Automation of Software Test (AST 2024)*, AST '24, page 5 pages, New York, NY, USA, apr 2024. ACM. doi: 10.1145/3644032.3644461.

[93] Andy Zaidman, Bart van Rompaey, Arie van Deursen, and Serge Demeyer. Studying the co-evolution of production and test code in open source and industrial developer test processes through repository mining. *Empirical Software Engineering*, 16(3): 325–364, 2011. doi: 10.1007/s10664-010-9143-7.