Towards a Software Greenability Model

Kirsten Gericke
kirstykromhout7@gmail.com

August 7, 2024, 96 pages

Academic supervisor: Ana Oprescu, a.m.oprescu@uva.nl
Daily supervisors: Chushu Gao, chushu.gao@softwareimprovementgroup.com
Pepijn van de Kamp, p.vandekamp@sig.eu
Host organisation/Research group: Software Improvement Group, www.softwareimprovementgroup.com

UNIVERSITEIT VAN AMSTERDAM
FACULTEIT DER NATUURWETENSCHAPPEN, WISKUNDE EN INFORMATICA
MASTER SOFTWARE ENGINEERING
http://www.software-engineering-amsterdam.nl

Abstract

As the Information and Communication Technology (ICT) industry continues to expand, its energy footprint has become a significant contributor to global carbon emissions. In response to this challenge, this thesis explores the creation and validation of a Software Greenability Model. "Greenability" refers to the ability of software to become more sustainable through the ease of implementing green software refactorings that reduce software energy consumption. While existing approaches focus primarily on direct energy consumption measurements, this model aims to measure and improve the environmental sustainability of software systems by integrating green software patterns and quality attributes defined by ISO standards. The relationship between software Greenability and refactoring effort was analysed by applying six code refactorings to nine open-source Java projects, as well as surveying 25 software developers to gather insights on the perceived effort for various refactoring tasks. By combining these subjective perspectives with quantitative data, the results demonstrate that, on average, all refactorings are easier to implement if Software Greenability is improved. The application of the Greenability model was shown to yield actionable insights and recommendations, resulting in a valuable resource for the software engineering community to promote sustainable software practices.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

As the Information and Communication Technology (ICT) industry continues to expand, its energy footprint has become a significant contributor to global carbon emissions. This presents the societal and economic challenge of how to balance the increasing demand for digital services with the urgent need to reduce energy consumption and mitigate climate change. It is clear that sustainability is gaining importance worldwide, as is the role of software in implementing sustainable processes and contributing to a sustainable society [1]. By developing a model for software "Greenability", this study aims to provide a practical tool for the software industry to assess and improve the energy efficiency of their products, not only through energy consumption or carbon emissions measurements but through the improvement of software quality attributes.

This approach also addresses economic challenges by potentially reducing operational costs associated with energy consumption in data centres and user devices. As governments and organisations worldwide increasingly prioritize green initiatives (for example, with the introduction of the Corporate Sustainability Reporting Directive, or CSRD), the ability to develop and maintain energy-efficient software could become a competitive advantage in the tech industry, encouraging innovation and economic growth while having a positive impact on the environment.

Consider the following scenario: A website receives an average daily traffic of 100 unique visitors, each engaging in multiple page views and refreshes. This website is hosted on cloud infrastructure, typically maintained by a large-scale technology company operating multiple data centres. These data centres contain servers and storage devices, all of which consume electricity with every interaction. When we expand this seemingly small user base across internet usage worldwide, the cumulative effect becomes obvious. Each page load, each data transfer, and each computation performed in response to user requests contributes to the overall energy consumption of software products. This energy usage directly translates to carbon emissions, particularly in regions where the electrical grid relies on fossil fuels.

This example illustrates the often-overlooked connection between software and environmental impact. It demonstrates that even small-scale digital interactions can have significant impacts on software energy consumption. As our reliance on software continues to grow, understanding these effects becomes increasingly important for developing environmentally sustainable software.

Software quality attributes are crucial for Greenability as they directly influence the ease with which software can be refactored to improve energy efficiency. If software is easier to refactor and easier to make sustainable, it becomes more realistic to address energy consumption issues at scale. It's impressive to consider that this major global challenge of reducing ICT's energy footprint can start to be addressed at a scale as small as individual lines of code, highlighting the positive impact software quality can have on both businesses and the environment.

1.1 Problem statement

The goal of this project is to develop a model for software "Greenability". This defines the ability of software to become more sustainable, in terms of energy consumption, in the future. This model effectively aggregates and weighs software quality attributes into a combined score. As a result, developers gain valuable insights into the most important software properties that impact refactoring efforts and identify opportunities for optimising energy efficiency.

The fundamental principle of "Greenability" is that software should not only be energy efficient but also energy adaptive (able to adapt their behaviour depending on energy concerns) [2]. "Greenability" is calculated by considering the level of software quality of various properties, for example, maintainability and reliability,

rather than directly measuring energy consumption. This model proves that better software quality reduces the effort needed to implement green software refactorings. The Greenability Model contains concrete definitions based on ISO standards, does not use overly exhaustive instrumentation but rather practical static source code analysis, and is applicable across a wide spectrum of frameworks, languages and platforms.

Sustainability is a widely used term and generally refers to the capacity of something to last a long time [1]. However, sustainability in software does not officially have one widely agreed-upon definition. Naumann *et al.* [3] defines **Green and Sustainable Software** as *"software, whose direct and indirect negative impacts on the economy, society, human beings, and environment that result from development, deployment, and usage of the software are minimal and/or which has a positive effect on sustainable development"*. Similarly to Naumann *et al.* [3], this project uses the terms 'green' and 'sustainable' interchangeably.

The development of this model is expected to contribute significantly to the broader field of sustainable software engineering by expanding existing research on software energy consumption, as well as aid the Software Improvement Group (SIG) in assessing and improving the "Greenability" of their and their client's software systems. At SIG, this problem is manifested in the form of outdated models (for example "A Practical Model for Evaluating the Energy Efficiency of Software Applications" [4]) and insufficient knowledge on improving the energy efficiency of their client's software, particularly regarding how this relates to SIG's current Software Quality Models. By considering state-of-the-art research, the proposed model aims to bridge this gap, offering a practical, adaptable, and comprehensive tool for evaluating software sustainability potential.

1.1.1 Research questions

The primary research question is: **How to define an adequate model of Software Greenability?** This is broken down into the following:

- **RQ1:** Which software quality properties determine software "Greenability"?
- **RQ2:** Which software quality metrics determine software "Greenability"?
- **RQ3:** How does the predictive accuracy of the model compare to developer's perceived refactoring effort?
- **RQ4:** How can the application of the model yield actionable, specific insights and recommendations to enhance software practices?

1.1.2 Research method

The research methodology for this project includes a comprehensive literature review and two distinct validation approaches. The literature review extensively surveyed existing software quality literature, focusing on ISO standards, industry practices, and academic papers to identify software quality properties, sub-characteristics, and metrics relevant to Greenability. This provided a theoretical foundation for the development of the Greenability model. To validate the model, two experiments were implemented. The first validation involved a correlation analysis between the identified Greenability metrics and the effort required for various code refactorings, measured through churn values (lines of code changed while implementing a refactoring). This approach allowed for analysis of the relationship between the proposed Greenability properties and metrics, and actual refactoring effort. In addition to this, a second validation was conducted through a survey distributed to software engineers, developers, and technical consultants. This survey gathered insights on the perceived effort required to implement code refactorings based on different software quality attributes, providing a practical, experience-based perspective on Greenability properties and metrics. By combining these three methodological approaches - literature review, correlation analysis, and survey - this study aimed to create a comprehensive validation of the Greenability model, ensuring it is both theoretically sound and actionable.

1.2 Contributions

This research makes the following contributions:

1. A list of properties, sub-characteristics and metrics that contribute to software "Greenability".
2. Validation of the Greenability model through correlation analysis between Greenability properties and metrics, and refactoring effort.
3. Validation through a survey of software professionals, assessing perceived effort for refactoring tasks based on different Greenability property and metric scores.
4. Comparison of measured and perceived effort for software refactorings.

5. A validated Greenability model which yields actionable insights and recommendations to promote sustainable software practices.

Necessary files and results can be found on this project's Github repository [1].

1.3 Outline

Chapter 2 introduces related literature and background concepts crucial for understanding the Software Greenability Model.

Chapter 3 describes the process of designing and analysing the model, with respect to selected properties and metrics.

Chapter 4 describes the methodology behind two validations of the model: correlational analysis and a survey. The former involves measuring various Greenability metrics of software systems, as well as the effort taken to implement various code refactorings, and determining the relationship between the two. This validation proves that improving software quality also reduces the effort needed to implement Green Software Patterns.

The results from these measurements, the survey, as well as statistical tests are presented in Chapter 5. These results are interpreted in Chapter 6, as well as discussing how they contribute to answering the research questions and threats to validity.

Chapter 7 discusses related work in the field of software quality and sustainability. The report is concluded in Chapter 8 with a summary of the research questions and future work.

[1] `https://github.com/KirstyGericke/Greenability-Thesis`

Chapter 2

Background

This chapter presents the necessary background information for this thesis. As we set out to build a Software Greenability Model, we need to prepare several key ingredients: Section 2.1 introduces the larger concept of Software Sustainability and how research in the field has evolved over time, as well as defines Green Software Patterns. Section 2.2 explains the legal context and thus the importance of the research conducted in this thesis. Section 2.3 explains what the ISO standards are, and which were used to justify the selection of properties and metrics for the Greenability model. Section 2.4 describes an industry static source code analysis tool that we use to measure the selected properties and metrics. As we measure Greenability as the effort of refactorings required by the Green Software Practices, Section 2.5 explains why we have selected to *churn* as the value to represent *effort*, which is measured by the unit *person months*.

2.1 Software Sustainability

Initially, sustainability in software was largely overlooked. Early software development mainly focused on performance, functionality, and cost. However, as the environmental impact of technology became more recognised, the concept of green software engineering aimed to reduce energy consumption and environmental footprint of software systems.

Recent studies have been establishing benchmarks for sustainable software products [5–7], and exploring ways to measure and reduce the energy consumption of software, discovering that efficient algorithms could significantly extend the battery life of mobile and embedded systems and reduce overall energy use [8]. Calero *et al.* [1] provides an overview of software sustainability by discussing how it can be integrated into software engineering practices through various frameworks, tools and methodologies. They discuss sustainable design principles, energy profiling, and the role of software engineers in reducing the environmental impact of technology throughout the software life cycle. Educational institutions have also started integrating sustainability into their curriculum, recognizing the importance of training future software engineers to consider sustainability in their work [1].

Dick and Naumann [9] defined sustainable software as *"software that minimizes its negative impacts on the environment, economy, and society throughout its life cycle"*. This definition highlighted the importance of considering sustainability from the beginning of the software development process, as well as its three primary dimensions: human, economic, and environmental. These three dimensions are mentioned in almost all literature sources in this domain, and are visualised in Figure 2.1.

- **Human Sustainability:** This dimension focuses on how software development and maintenance impact sociological and psychological aspects of the software development community. It includes labour rights, psychological health, social support, social equity, and liveability within software development environments.
- **Economic Sustainability:** This dimension focuses on software life cycle processes and how they protect stakeholders' investments, ensure benefits, reduce risks, and maintain assets. It addresses the financial aspects of software sustainability, aiming to balance cost efficiency with long-term viability.
- **Environmental Sustainability:** Often referred to as "Green Software", this dimension focuses on how development, maintenance, and usage of software products affect energy consumption. This can be divided into into:
  - **Green IN Software:** when the environmental issues are related to software itself. This includes

practices of designing, developing, and maintaining software products to minimize their negative impact on the environment, such as improving energy efficiency and reducing resource consumption during the software's life cycle.

– **Green BY Software:** when software is used as a tool to support sustainability goals in any domain. This includes developing software applications that help monitor, manage, and reduce the environmental footprint of hardware systems and industrial processes.

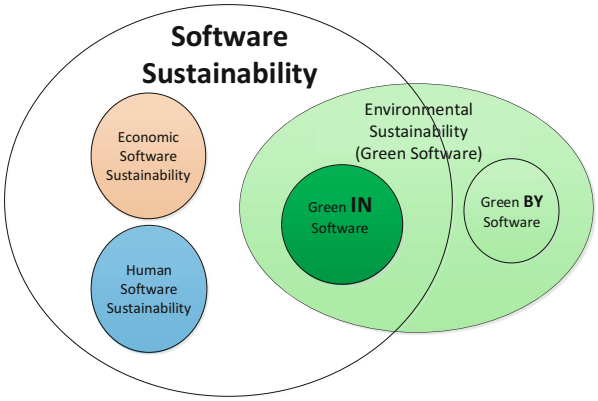


Figure 2.1: Software Sustainability Dimensions [1]

One of the most significant Software Sustainability models that has been developed is The GREENSOFT model [3], which presents a quality model for green and sustainable software. This model takes into account the life cycle of software products (first, second and third-order effects), sustainability criteria and metrics, procedure models and how to provide recommendations for areas of improvement. They specify three categories of sustainability criteria and metrics for software products: Common Quality Criteria and Metrics (ISO standards [10]), Directly Related Criteria and Metrics, and Indirectly Related Criteria and Metrics. The one disadvantage of this model however is they lack explicit definitions of metrics that can be used to measure energy consumption. Further exploration of various metrics for this purpose is discussed in Related Work in Section 7.3.

Another important previous study relevant to Greenability is a paper titled "A Practical Model for Evaluating the Energy Efficiency of Software Applications" by Kalaitzoglou *et al.* [4]. This model uses the Goal-Question-Metric (GQM) approach to identify 8 metrics used to answer 8 research questions on three levels (Energy behaviour, Capacity and Resource Utilisation). This paper was written in collaboration with the University of Amsterdam and presents the ME^{3}SA model to measure various aspects of software energy usage. Although this research was very significant and relevant, SIG recognises that more than 10 years have passed since this publication and the application of such a model proved more difficult than expected. In particular, the need for the collection of runtime utilisation data is often a big obstacle. This paper addresses this issue by defining Greenability in terms of software quality metrics, where dynamic energy measurements are not required, yet conclusions on the system sustainability can still be observed.

2.1.1 Green Software Patterns

Green Patterns are design and coding practices aimed at improving the energy efficiency of software. These are categorized at source code level (eg. use ArrayMap instead of HashMap) and design level (eg. opt for dark colours when designing the UI). Identifying patterns at the code level is useful as they can be transformed into a more energy-efficient alternative - known as *refactoring*.

Feitosa *et al.* [11] demonstrates how patterns with various scopes can help build energy-efficient software. Some patterns address energy consumption as the main concern, while others have an energy-related side effect. Feitosa *et al.* discusses 32 Green software patterns, 8 on the code level and 24 on the design level. Since it is not feasible to present all 32 patterns, one example is described: `DrawAllocation`

`DrawAllocation` *"occurs when new objects are allocated along withdraw operations, which are very sensitive to performance. In other words, it is a bad design practice to create objects inside the onDraw method of a class which extends a View Android component."*

The recommended alternative is to move the allocation of independent objects outside the method, turning it into a static variable.

```
public class CMView extends View{
    @Override
    protected void onDraw(Canvas C){
    RectF rectF1 = new RectF();
    ...
    If(!clockwise){
        rectF1.set(X2-r, Y2-r, X2+r, Y2+r);
        ...
}}}
```

Listing 2.1: Before

```
public class CMView extends View{
    RectF rectF1 = new RectF();
    @Override
    protected void onDraw(Canvas C){
    ...
    If(!clockwise){
        rectF1.set(X2-r, Y2-r, X2+r, Y2+r);
        ...
}}}
```

Listing 2.2: After

As seen from this example, the refactoring implemented turns a local variable into a field. This refactoring is equivalent to a refactoring used in Sahin *et al.* [8], namely `Convert Local Variable to Field` which is defined as *"creates a new field by turning a local variable into a field."*. This is an important observation as the data set created by Sahin *et al.* [8] is used in this project during the validation step. This paper implements 6 most common code refactorings on 9 software systems, and is described in detail in Section 4.2. This example proves that the most common code refactorings are often found in Green Software Patterns, and further justifies the choice of using Sahin *et al.* [8] for the Greenability Model's validation. Full descriptions of this process is explained in this report, so it is not necessary to explain the importance in this section, but keep in mind it will become important later in this report.

2.2 Corporate Sustainability Reporting Directive

The Corporate Sustainability Reporting Directive (CSRD) is a new European Union (EU) legislation that requires all large companies and listed companies to publish regular reports on the social and environmental risks they face, and on how their activities impact people and the environment. Some non-EU companies will also have to report if they generate over EUR 150 million on the EU market. More information can be found on their website[1].

With this introduction of CSRD, many more companies will be encouraged to invest time and effort into improving not only the sustainability of their company as a whole, but specifically of their software. This aids in reducing the overlooked effect software has on the sustainability of companies. This legislation supports the importance of sustainable software practices and the growing legal and societal expectations for companies to report on their sustainability efforts. The CSRD emphasises transparency in sustainability reporting, aligning with the goals of Greenability by promoting practices that lead to measurable improvements. It also aligns with Greenability in the context of providing a structured approach to implement these measurements. The CSRD shows the practical applicability of the Greenability Model in helping companies accurately report on these requirements.

2.3 ISO Standards for Software Quality Attributes

Three prominent ISO standards define software quality measurement: ISO 9126, ISO 25010 and ISO 5055. These standards contribute significantly to the Greenability model created in this project for the concepts used as well as the method of measuring these concepts. These standards have been derived from other well-known quality models. Software vendors pay a considerable amount of money to obtain an ISO certification. These standards are widely accepted both by industrial experts and academic researchers due to the standardisation process.

ISO 9126 and ISO 25010

Software quality is a measure of how well software adheres to design and fulfils non-functional requirements, such as security and maintainability. It involves quantifying the degree to which software meets various *quality attributes*, which help developers determine how well the software performs and maintains its quality over time [12]. Key aspects of software quality include:

- Testability: The software can be easily tested to ensure it functions correctly.
- Comprehensibility: The software is straightforward to understand and follow.

[1] `https://finance.ec.europa.eu/capital-markets-union-and-financial-markets/company-reporting-and-auditing/company-reporting/corporate-sustainability-reporting_en`

- Modifiability: The software can be edited and upgraded easily without introducing new errors.

It is important to make the distinction between ISO 9126 [13] and ISO 25010 [10]. ISO 25010, which was published in 2011, superseded ISO 9126 published in 2001. The main difference between the two lies in how they categorize and define non-functional software quality requirements. ISO 25010 added two additional product quality characteristics to the six specified in ISO 9126: security and compatibility [12]. Figure 2.2 and Figure 2.3 highlight these differences.

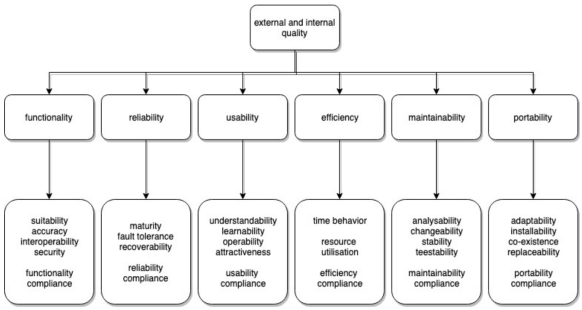


Figure 2.2: ISO/IEC 9126 categorization of software quality requirements.

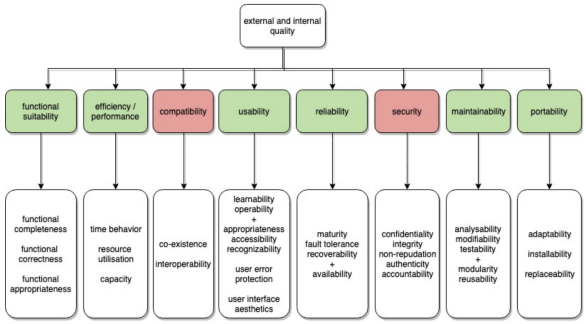


Figure 2.3: ISO/IEC 25010 categorization of software quality requirements [10]

ISO 25010, titled "Systems and software engineering – Systems and software Quality Requirements and Evaluation (SQuaRE) – System and software quality models", is a software quality standard. It describes models, including characteristics and sub-characteristics, for software product quality, and software quality in use [12].

ISO 5055 and CWE

ISO/IEC 5055:2021 is a standard for measuring the internal structure of a software product focusing on four business-critical factors: Security, Reliability, Performance Efficiency, and Maintainability [14]. ISO 5055 fills a gap in previous standards by providing measures to detect severe structural weaknesses in software during development, preventing operational problems before they occur. The standard utilises the Common Weakness Enumeration (CWE) list to identify the most critical software weaknesses.

CWE stands for Common Weakness Enumeration and is a community-developed list of common software and hardware weaknesses. According to the CWE website [2], *a "weakness" is a condition in a software, firmware, hardware, or service component that, under certain circumstances, could contribute to the introduction of vulnerabilities.* The CWE List and associated classification taxonomy identify and describe weaknesses in terms of CWEs.

In practical terms, implementing ISO 5055 involves using static analysis tools to detect and report these weaknesses across the entire technology stack. This ensures that the software is evaluated in its entirety, considering all inter-dependencies, and helps organisations maintain high standards of software quality and security [14]. This is clearly very relevant to this project considering measurements are required to be imple-

[2] More information about CWE can be found at `https://cwe.mitre.org/about/index.html`

mented using static analysis tools, and this ISO standard is strongly related to the previous ISO standards 9126 and 25010.

2.3.1 Clarification of non-ISO terms

The first clarification that needs to be made, is that the term *quality attribute* represents the same concept in this report as the term *property*. There are two software quality attributes, or properties, used in this thesis which are not defined by ISO, and do not have a consistent definition over literature sources: *understandabilty* and *greenability*.

Understandability

Understandability is a term which can be viewed from two different perspectives: how well users understand a software application, or how well developers understand a code base.

ISO 9126 [13] defines understandability as the *capability of the software product to enable the user to understand whether the software is suitable and how it can be used for particular tasks and conditions of use*. This definition emphasises the importance of making software comprehensible to users, which in turn facilitates ease of use and effective implementation. For the scope of this project, this perspective is not relevant to Greenability. Instead, the alternative view is explored.

Lavazza *et al.* [15] defines software Understandability, in the context of software engineering, as the degree to which software code can be comprehended by developers. This comprehension is crucial for maintaining and evolving software. The measurement of understandability can involve various factors such as the time taken to understand code, the correctness of understanding tasks, and subjective ratings by developers.

This distinction is mentioned in Chapter 3 when analysing which properties and metrics are relevant to Greenability, and it is clarified that understandability is applicable in the context of developers, rather than users, understanding code.

Greenability

This project defines the term Greenability as the ability of software to become more sustainable in the future, mainly centred around the ease at which green software refactorings can be implemented.

This is not a standardised term and is not very commonly used in current literature, apart from one exception: Calero *et al.* [16]. This paper defines the term "Greenability" in relation to the ISO 25010 standard, focusing on environmental sustainability on the first environmental impact level. They define four sub-characteristics of product greenability as energy efficiency, resource optimisation, capacity optimisation and perdurability. They further add greenability to the "quality in use" model by first determining three sub-characteristics (efficiency optimisation, user's environmental perception and minimization of environmental effects), followed by defining **Greenability in use** as *Degree to which a software product can be used by optimizing its efficiency, by minimizing environmental effects and by improving the environmental user perception*. They use the term "greenability" in a much wider scope than this thesis considers.

Calero *et al.* [16] provides valuable research in the software sustainability domain, however, the differences in definitions for Greenability are not seen as a major conflict, and perhaps when further research is done in these areas, a standardised term can be clarified. For the purpose of this project, Greenability suits the context as suffix *-ability* aligns with similar software quality attributes, such as maintain*ability*, reli*ability*, analys*ability*, and many others.

2.4 Sigrid: An industry static source code analysis tool

This project was created in collaboration with the Software Improvement Group (SIG), and makes use of their static source code analysis tool: Sigrid. Sigrid is a tool provided by SIG to their clients, measuring the quality of their software using various quality models. For example, they measure software maintainability, reliability, performance, security, and more. These models are based on ISO standards, and report results on a 5-star rating scale based on benchmarked data from thousands of software systems. This provides a comparable and reliable perspective of software quality and allows the clients of SIG to view how their system compares to market standards.

Sigrid communicates results as bench-marked star ratings. The 5-star scale can be visualised in Figure 2.4. As seen in this figure, 3 stars represent the equivalent quality of 30% of industry systems. In reality, the scale ranges from 0.5 to 5.5, and it can be seen how these values round up or down to get the number of stars for

that score. In many cases, 5 stars is unrealistic to achieve, unless the systems are incredibly small. All metrics used in this paper follow this star scale to represent a score or rating. This allows all metrics and properties to be comparable and capable of identifying the most essential areas for improvement.

Figure 2.4: 5-star rating scale used by Sigrid.

Essentially, Sigrid's insights are based on source code analysis. Analyses are run on the code "as is", so without actually running the system. This is known as *static analysis*, as opposed to *dynamic analysis* [17]. Dynamic analysis of software is more typically done by the developers themselves, as it requires a simulation of how a system will behave in operation. The static analysis methods used by Sigrid are based on the ISO 5055. There are pros and cons of each of these approaches. Dynamic analysis could provide a more accurate representation of certain properties, for example, *test coverage* arguably provides a better understanding of *testability* than *test code ratio* would. On the other hand, static analysis is a faster and easier analysis method, and when metrics are standardized and their validity is scientifically supported, the pros outweigh the cons, especially when it comes to a company like SIG which analyses thousands of systems in extensive detail.

2.5 Effort, Churn and Person Months

Calculating effort for software development is important for project management, estimating costs of scheduling and planning, and evaluating business performance. It helps managers allocate resources effectively, predict timelines, and ensure that projects stay within budget. The main challenge in measuring effort is its subjectivity. The amount of effort required for the same task can vary significantly between developers based on their skills, experience, and working styles. For example, an experienced developer could complete a task quicker and with less cognitive load compared to a less experienced developer.

Effort in software development can be interpreted as time taken to complete a task, cognitive load required, or measurable values such as lines of code (LOC) changed. An ideal method for measuring effort might involve a controlled experiment where multiple developers refactor the same piece of code, their time is measured, and their cognitive load is assessed through a questionnaire. This, however, is difficult to implement on a large scale considering the experimental conditions required, as well as the impact individual performance has on results. Practically, this approach may have too many confounding variables.

Using LOC changed as a metric for effort is a practical alternative. In this project, we use the term *churn* to represent this concept. LOC is easy to measure across many systems and provides an objective measure of effort that is independent of the individual performing the task. This also holds true regardless of whether refactorings are implemented manually by developers or automatically by tools.

By focusing on LOC as a measure of effort, this project aims to provide a standardised approach to evaluating the Greenability of software systems. This approach allows for consistent comparisons across different systems and refactorings, supporting the goal of improving software sustainability through more efficient refactoring practices.

A common model for measuring this effort is The Constructive Cost Model, or COCOMO model, which calculates the Person Months needed for a software system. This model was developed by Barry Boehm in 1981, and provides estimates for project cost, effort, and schedule by assessing various parameters. The model has three types: Basic, Intermediate and Detailed COCOMO.

The basic model uses a simple formula for predicting how many Person Months of work are required based on the size of the project:

$$E = a * (KLOC)^b PM \tag{2.1}$$

$$Tdev = c * (E)^d \tag{2.2}$$

where E is effort applied in Person-Months, $KLOC$ is the estimated size of the software product indicated in Kilo Lines of Code, $Tdev$ is the development time in months, and a, b, c are constants determined by the category of software project (organic, semi-detached and embedded). In reality, no system's effort and schedule can be solely calculated based on Lines of Code. The Intermediate model considered 15 Cost Drivers, or multipliers, such as product, hardware, personal and project attributes. These values result in *The Effort Adjustment Factor (EAF)*, which is determined by multiplying the effort multipliers associated with each of the 15 attributes. Therefore Equation 2.1 is updated to:

$$E = a * (KLOC)^b * EAF\ PM \tag{2.3}$$

Finally, the Detailed COCOMO Model dives even deeper into project-specific factors, such as team experience, development practices and software complexity, to provide a more accurate estimation of effort specific to the project being considered.

The tool Sigrid automatically converts the lines of code into Person Months and outputs both LOC and PM values. These values are used to validate the Greenability model in this project, by measuring the churn values of various code refactorings.

Chapter 3

Designing a Software Greenability Model

A comprehensive analysis of existing software quality literature was conducted to identify software quality properties, sub-characteristics and metrics that may impact software Greenability. This step was crucial in establishing a theoretical basis for Greenability. This aligns with the systematic approach used in exploratory research, as described by Kitchenham and Charters [18], where criteria are selected based on relevance to the research goals.

To design a Greenability Model consisting of properties, sub-properties and metrics, current literature was analysed by exploring software quality attributes, their definitions, how they are calculated, the effect they have on software and their relationship with other quality attributes. This informed not only whether or not they would be applicable to Greenability, but how they impacted Greenability.

Important sources for Greenability include ISO standard documentation [10, 12–14], research on current tools and models related to Software Sustainability (Greensoft [3], ME^{3}SA [4]), documentation and previous studies conducted by SIG [19–21], including Sigrid [17], and research conducted by The Green Software Foundation [22–24]. The textbook *Software Sustainability* [1, 11, 25, 26] contains a comprehensive collection of scientific research covering the most significant aspects of software sustainability. Zotero was utilised as a reference manager for the literature review, and additional sources were collected using snowballing from IEEE Xplore, SpringerLink and ACM digital library.

The terms used in this model follow a hierarchical structure, where *properties* contain multiple *sub- characteristics*, and these are measured by various *metrics*. This is visualised in Figure 3.1. There is not always a one-to-one relationship between metrics and sub-characteristics, but rather a combination of various metrics that contributed to one sub-characteristic, while a different subset of metrics, allowing for overlaps, contributed to another. For example, the metric *duplication* is used to calculate both *analysability* and *modifiability*, and the metric *volume* is used to calculate both *analysability* and *testability*. While these concepts and their definitions are standardised, the terms used to describe them are not. For example, Heitlager *et al.* [27] uses the term *property* for what this project describes as a *metric*. Similarly, a *property* in this project represents the same concept as the term *quality attribute*. Regardless of the terms used, this hierarchical structure of 3 layers is common throughout the literature.

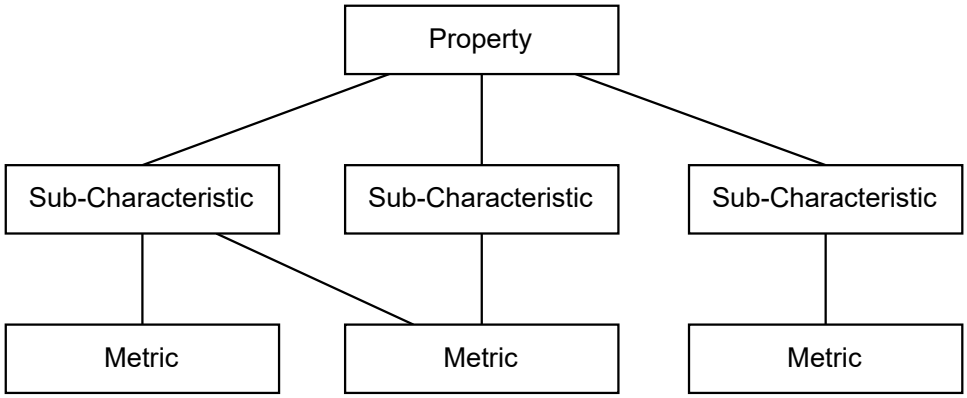


Figure 3.1: Hierarchical Structure of Properties, Sub-Characteristics and Metrics

These properties, sub-properties and metrics were selected based on their alignment with research goals and the definition of Greenability. Only those metrics and properties that can be accurately and reliably measured were selected to ensure the validity and reliability of the model. It was also important for them to be applicable across different types of software and contexts to ensure the generalisability of the model. Preference was given to concepts that comply with recognized standards, such as ISO/IEC standards, to ensure the research is supported by widely accepted frameworks.

The process involved reviewing literature sources exploring software sustainability and which properties have been the most widely recognized and validated as the most critical for developers to consider when developing software. Once the applicable properties were selected, their sub-characteristics were analysed. By using standardised definitions of properties, their sub-characteristics were often mentioned in their definitions. For example, a source might state "a system is *maintainable* - meaning it is *analysable* and *testable*". To be as thorough as possible, all sub-characteristics were also analysed with regard to their applicability to Greenability.

If a property is applicable to Greenability, it does not automatically mean all sub-characteristics are as well. For example, only two out of the four sub-characteristics of *portability* were applicable to Greenability. Similarly, not all metrics of a property were applicable to Greenability just because the property is applicable. For example, *Knowledge Distribution* was not applicable to Greenability, as the distribution of knowledge within a team of developers does not impact the structure of the source code, while other metrics for *Architecture Quality* were applicable.

Section 3.1 discusses properties relevant to Greenability across three of the most useful literature sources. From this analysis, seven property selections are made that contribute to Greenability. Each of these properties is associated with several sub-characteristics. For example, *maintainability* is a *property* and consists of the *sub-characteristics analysability, changeability, stability* and *testability*. Each of these is discussed in detail on how they impact Greenability.

The final analysis to perform while selecting concepts for the Greenability model was the individual metrics. The same logical analysis was performed on all metrics and explained in detail while providing references to published and reliable sources to justify the selections made.

Section 3.2 further explores the seven properties selected, and determines metrics which need to be measured to calculate each property. Once again, each topic discussed relates to how a metric impacts Greenability.

Section 3.3 summarises all property and metric selections in a diagram which defines Greenability. The final model for Greenability is presented, however, it is clear that although justified by literature and reliable sources, further validation of these selections needed to be conducted to improve the validity and gain additional perspectives on the impacts these properties and metrics had on the ease of refactoring systems to improve energy consumption. These validations were performed using two separate research methods: correlation analysis and a survey provided to software engineers, developers and consultants. These methods are discussed in Chapter 4.

3.1 Analysis of Greenability Properties

Table 3.1 presents a concept matrix for most of the properties investigated for this section. It is clear that the list of software quality attributes and properties that exist is non-exhaustive. The continuous growth in this research area results in the criteria for evaluating software quality being dynamic and subject to expansion. For this reason, it is impractical to report on all properties unrelated to software energy consumption and Greenability. Instead, this section summarises results from three sources which have provided the relevant basis for understanding the key properties of Greenability. The most significant sources of information for software quality properties were ISO standards [13][10], Albertao *et al.* [28], and the documentation for the Software Improvement Group's tool, Sigrid [17]. This subsection is structured according to these three sources, discussing each property and stating clearly when a property selection is made that contributes to the Greenability Model.

Concept	[10, 13]	[17]	[27]	[28, 29]	[3]	[30]	[15]	[31, 32]	[33]	[34]	[35]	[16]
Maintainability	✓	✓	✓									
Reliability	✓	✓	✓									
Modifiability	✓			✓	✓							✓
Reusability	✓			✓	✓							✓
Portability	✓		✓	✓	✓			✓				✓
Functionality	✓											
Security	✓	✓										
Supportability				✓	✓						✓	
Performance	✓	✓	✓	✓	✓							✓
Dependability				✓	✓						✓	
Usability	✓		✓	✓	✓	✓					✓	✓
Accessability	✓			✓	✓						✓	✓
Predictability				✓	✓							
Efficiency	✓	✓	✓	✓	✓							✓
Footprint				✓	✓							
Testability	✓	✓	✓	✓					✓	✓	✓	✓
Understandability	✓		✓	✓			✓					
Scalability											✓	
Modularity	✓	✓									✓	✓
Greenability												✓
Open Source Health		✓										
Cloud Readiness		✓										
Architecture Quality		✓										

Table 3.1: Non-exhaustive list of Software Quality Properties.

Source 1/3: ISO standards

The international standards for software quality, namely ISO/IEC 9126 [13] and ISO/IEC 25010 [10], outline a model composed of six characteristics and sub-characteristics for both software product quality and software quality in use, as seen in Figure 3.2. This framework provides a foundational reference for the literature survey by identifying key characteristics essential for developers during the software system development process.

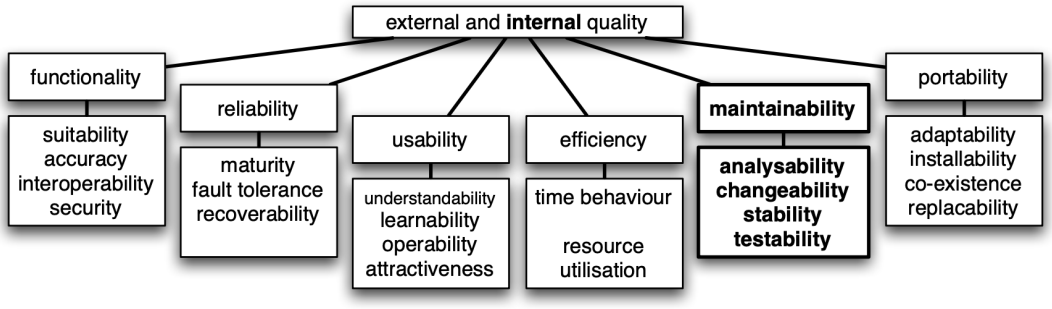


Figure 3.2: Breakdown of the notions of internal and external software product quality into 6 main characteristics and 27 sub-characteristics [27].

Each of these six properties was analysed in detail to determine their relevance to Greenability. After each property analysis, it is stated which properties are selected for the Software Greenability Model. Through this comprehensive analysis, a subset of properties was identified as particularly applicable to reducing the effort taken in software refactorings. By focusing on these key properties, developers can better target their efforts to enhance the Greenability of their systems.

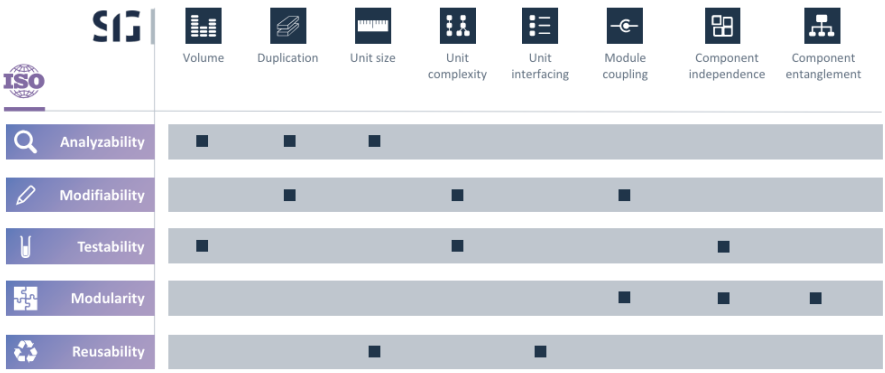


Figure 3.3: Maintainability sub-characteristics and metrics [19]

1. *Maintainability* represents many applicable aspects of Greenability. A highly maintainable system — meaning it is analysable, changeable, stable, and testable — facilitates easier implementation of green software patterns and refactorings. Vis *et al.* [19] defines several sub-characteristics and metrics used to measure maintainability, as seen in Figure 3.3. The sub-characteristics in this model are based on the ISO 9126 [13], while the metrics used to measure these were selected by Vis *et al.* [19]. This classification was created based on the ISO 9126, before the ISO 25010 update was released. Each sub-characteristic of maintainability mentioned in both ISO 9126 and ISO 25010 are defined as follows:

    - **Analysability:** Degree of effectiveness and efficiency with which it is possible to assess the impact on a system of an intended change to one or more of its parts, to diagnose a system for deficiencies or causes of failures, or to identify parts to be modified. High Analysability allows developers to quickly understand the code, identify areas for improvement, and plan sustainable modifications.
    - **Changeability** ensures that the code can be easily modified to incorporate new, more efficient algorithms and practices. This improves Greenability by allowing the system to adapt to more sustainable practices with minimal effort.
    - **Stability** implies that changes can be made with minimal risk of introducing new issues, ensuring that sustainable practices can be implemented without compromising the system's reliability. This enhances Greenability by making the system more robust and dependable during sustainability-focused refactorings.
    - **Testability:** Degree of effectiveness and efficiency with which test criteria can be established for a system and tests can be performed to determine whether those criteria have been met. Testability not only allows for energy consumption measurements to be taken[1], but it ensures when refactorings take place that implement sustainability patterns, there is a lower risk of introducing new bugs or changing the system's functionality.
    - **Modifiability:** Degree to which a system is composed of discrete components such that a change to one component has minimal impact on other components. This improves Greenability by making it easier to implement and manage sustainable practices without causing widespread issues.
    - **Reusability:** Degree to which a product can be used as an asset in more than one system, or in building other assets. This improves Greenability by reducing the need to create new components from scratch, therefore saving resources and promoting sustainable development practices.
    - **Modularity:** Degree to which a system can be effectively and efficiently modified without introducing defects or degrading existing product quality. This enhances Greenability by allowing for the easy incorporation of green software practices through targeted modifications.

    Together, these sub-characteristics reduce the effort required for developers to refactor and enhance the software, making the system more adaptable to sustainable practices and improving its long-term Greenability. This property is implemented in detail in Heitlager *et al.* [27] through designing a software Maintainability Model, justifying the relevance and applicability of maintainability measurements to

[1] If a system has an in-depth testing suite, those tests imitate the system as if users interact with the system. This means developers can simply run the test files for a system and get energy consumption measurements as a result.

real-world industry systems. For these reasons, the first selection is made of a property contributing to Greenability:

> **Property Selection 1:** Software with higher Maintainability will require less effort to improve sustainability.

2. *Reliability* is very applicable to Greenability. A reliable system is characterised by the following sub-characteristics according to ISO standards [10, 13]:

   - **Maturity:** Capability of a software product to avoid failure as a result of faults in the software.
   - **Fault Tolerance:** Degree to which a system operates as intended despite the presence of hardware or software faults.
   - **Recoverability:** Degree to which, in the event of an interruption or a failure, a system can recover the data directly affected and re-establish the desired state of the system.
   - **Availability:** Degree to which a system is operational and accessible when required for use.

   All of these definitions relate in some way to software avoiding faults and failures. Reliable systems minimize the occurrence of unexpected issues during refactoring, allowing developers to focus on implementing green software practices without the constant need for debugging. This stability ensures that changes aimed at improving sustainability can be made with confidence, knowing that the system's functionality will remain unchanged. By ensuring an error-free environment, reliability ensures a system's ability to evolve sustainably, leading to a higher Greenability. For this reason, the second Greenability property selection is made:

   > **Property Selection 2:** Software with higher Reliability will require less effort to improve sustainability.

3. *Functionality* is not applicable to Greenability and does not influence the potential a system has to become more sustainable. The definition of functionality across ISO 9126 and 25010 differs significantly, primarily by moving the sub-characteristic of *security* to become a separate software quality on its own [10]. The newest classification of metrics for functionality includes:

   - **Functional Completeness:** Evaluates whether the software includes all the necessary functions specified in the requirements.
   - **Functional Correctness:** Measures if the software provides accurate results with the required precision.
   - **Functional Appropriateness:** Assesses how well the software functions facilitate the completion of specified tasks

   These metrics focus on ensuring that the software meets its specified requirements and performs its intended functions correctly and appropriately. While these aspects are critical for the software's operational effectiveness, they do not directly impact the effort required for refactoring the code to implement sustainable practices. Functional Completeness, Correctness, and Appropriateness are primarily concerned with the current capabilities and accuracy of the software, rather than its adaptability for future sustainability refactorings. For these reasons, this aspect is disregarded for inclusion in the Greenability model.
4. *Useability* determines whether specific users can use the application in specific conditions. According to Britton [12] and the ISO 9126 [13], each sub-characteristic has the following definition:

   - **Learnability:** Refers to how easy it is to learn how to use a product or system.
   - **Operability:** Refers to whether a product or system has attributes that make it easy to operate and control.
   - **Appropriateness and Recognizability:** Refers to how well you can recognize whether a product or system is appropriate for your needs.
   - **User Error Protection:** Refers to how well a system protects users against making errors.
   - **Attractiveness or User Interface Aesthetics:** Refers to whether a user interface is pleasing.
   - **Accessibility:** Refers to how well a product or system can be used with the widest range of characteristics and capabilities.

- **Understandability:** Refers to the capability of the software product to enable the user to understand whether the software is suitable and how it can be used for particular tasks and conditions of use.
- **Usability compliance:** Refers to the degree to which the software adheres to standards, guidelines, and regulations related to usability.

It is clear that these are defined in terms of how easily the user can use and understand the software product, rather than how easily the developer can use and understand the code base. This perspective is clearly not relevant to Greenability, however, it brings to light an alternative perspective on the concept of *understandability* which is very relevant to Greenability: how well developers understand code. Alternative sources from the literature have been found to use the following measurements for this perspective of understandability:

- **Task Execution Time:** Measures the time taken to carry out a task given a software source code [36–38], indicating how quickly developers can understand the code. Shorter understanding times reduce the effort required for implementing Green IT practices by making it easier and quicker for developers to comprehend and refactor the code.
- **Correctness:** Measures the degree of success of maintenance tasks [37, 38]. High correctness in understanding tasks ensures that sustainable changes can be implemented accurately, reducing the effort and risk of introducing errors during green refactorings.
- **Subjective Rating:** Measures the subjective opinion on an ordinal scale of how well developers believe they understand code [37, 39]. Subjective ratings can provide insights into code complexity and readability, but they are less precise than objective measures, making them only partially useful for determining the effort needed for green refactorings.
- **Physiological Measurements:** Measures activities occurring in the human body involving the brain and heart [40, 41], to gauge cognitive load and understandability. This provides indirect insights into understandability and does not directly impact the structural code or the effort required for implementing sustainable practices.

The specific metrics Lavazza *et al.* [15] uses to calculate understandability are described in detail in Section 3.2. Understandability is selected as another Greenability property:

> **Property Selection 3:** Software Understandability, in the context of developers understanding source code, is applicable to Greenability.

5. *Efficiency* is yet another term which seems to have multiple perspectives:

   - **Performance Efficiency:** Refers to the ability of the software to provide appropriate performance relative to the amount of resources used under stated conditions. This includes aspects such as response time, throughput, and scalability [10].
   - **Resource Efficiency:** Focuses on the optimal use of system resources such as CPU, memory, and energy. Efficient resource utilization ensures that the software can perform its functions without unnecessary wastage [42].
   - **Cost Efficiency:** Involves minimizing the costs associated with software development, maintenance, and operation. This perspective is important for ensuring that the software is not only effective but also economically sustainable [43].

   The ISO 25010 specifies the characteristic as *efficiency/performance*, hence, focusing on performance efficiency. For clarity, from this point on, this report will specify *Performance Efficiency* rather than *efficiency*.

   The metrics used to determine the ISO definition are time behaviour, resource utilization and capacity. By optimizing these metrics, performance efficiency ensures that software operates at peak performance with minimal waste of resources. Efficient use of CPU, memory, and other resources reduces the environmental impact of software operations. Furthermore, it is hypothesised that systems that perform efficiently require less frequent and less extensive maintenance, which minimizes the effort and resources needed for ongoing support and updates. This optimisation not only improves the performance of the system but also supports long-term sustainability by making the software more adaptable

to future green practices and technologies.

Although performance efficiency metrics are commonly determined using dynamic analysis methods [44], Sigrid utilises the methods described in the ISO 5055 [14] which utilises CWEs to measure metrics statically. Please refer to Section 2.3 for more information about ISO 5055 and CWEs. This model is applicable to Greenability considering this project specifically focuses on statically measured metrics, and this property aligns very clearly with software sustainability goals. Therefore Performance Efficiency is selected as a Greenability property:

> **Property Selection 4:** Software with higher Performance Efficiency will require less effort to improve sustainability.

6. *Portability* is defined as the ability of a system to run under different computing environments [28]. According to the ISO 25010 [10], portability is defined as containing four sub-characteristics: adaptability, install-ability, co-existence and replace-ability. These are defined by Britton [12] and ISO [10] as follows:

   - **Adaptability**: Refers to how well a product or system can be adapted for different or evolving hardware, software, or other usage environments. Adaptable software can be easily modified to take advantage of more energy-efficient hardware or configurations. For instance, if newer, more sustainable hardware becomes available, adaptable software can be adjusted to run efficiently on this hardware, therefore reducing overall energy consumption. Adaptability also makes it easier to incorporate green software patterns, such as energy-efficient algorithms or resource management strategies, without extensive reworking of the existing code base.
   - **Replaceability:** Refers to how well a product can replace another comparable product or the ability to easily substitute one part of the software with another, often without affecting the overall system. If certain components or modules can be replaced with more energy-efficient alternatives, replaceability ensures that these upgrades can be done easily, thereby improving the software's sustainability without major disruptions. Software designed with replaceability in mind is often also more modular, making it easier to isolate and improve specific parts of the system for better resource efficiency. Finally, replaceability supports the continuous evolution of software. As new green technologies and methodologies emerge, they can be integrated into the system by replacing outdated or less efficient components, improving Greenability.
   - **Co-existence:** Refers to the degree to which a product can perform its required functions efficiently while sharing a common environment and resources with other products, without having a detrimental impact on any other product. This sub-characteristic was subsequently moved under *Compatibility* in ISO 25010 and is therefore not considered when determining *Portability* for this project.
   - **Installability:** Refers to how successfully a product or system can be installed and/or uninstalled. Installability is not directly applicable to Greenability as it focuses more on the ease of deployment rather than ongoing sustainability improvements.

   Therefore only the Adaptability and Replaceability sub-characteristics are selected for the Greenability model, resulting in the following properties selection:

> **Property Selection 5:** Software with higher Portability (Adaptability and Replaceability) will require less effort to improve sustainability.

Source 2/3: Sigrid Software Quality Models

The model described in Heitlager *et al.* [27] was implemented using a Software Analysis Tool, Sigrid [17], developed by the company the Software Improvement Group. This tool is used later in this project during the validation step (Section 4.4), which justifies the importance of analysing the models and metrics it is capable of measuring. Sigrid not only implements the Maintainability Model based on ISO 9126, but multiple additional Software Quality Models including Security, Open Source Health, Reliability, Performance, Cloud Readiness and Software Architecture. For clarity, Sigrid refers to these concepts as *Qualities*, however for the remainder of this project, they will be referred to as *Properties*. Figure 3.4 is an example of how these seven qualities are displayed by this tool.

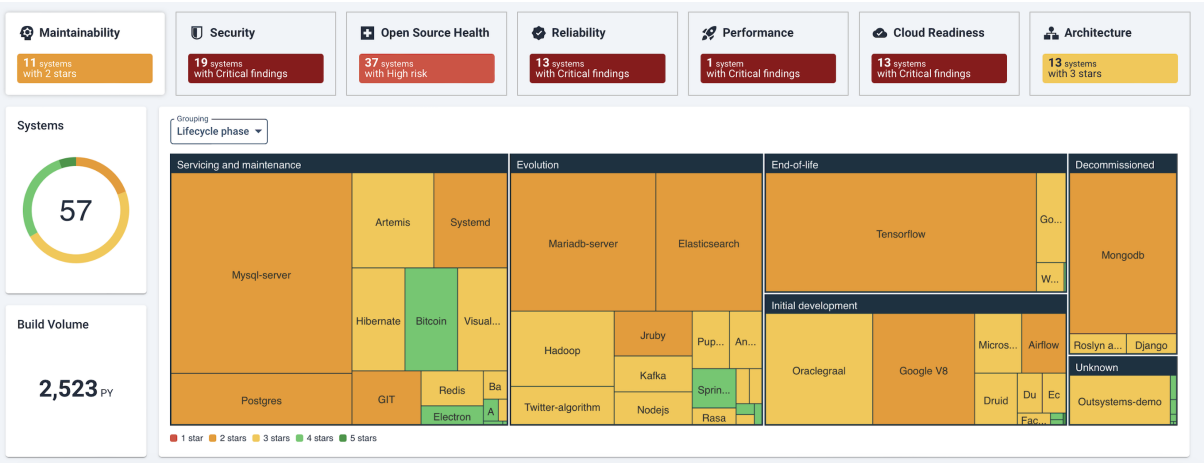


Figure 3.4: Example view of Sigrid Dashboard displaying 7 quality models: Maintainability, Security, Open Source Health, Reliability, Performance, Cloud Readiness and Architecture.

The definition and applicability of each of these seven properties are explained in the same manner as the previous section. Note there are some overlaps between Sigrid Qualities and ISO properties, and therefore in the case where a property selection has already been made, it is referenced and not stated a second time.

The seven properties are as follows:

1. *Maintainability* is accounted for in Property Selection 1.
2. *Reliability* is accounted for in Property Selection 2.
3. *Performance Efficiency* is accounted for in Property Selection 4.
4. *Security* is defined as *"The degree to which a system protects information and data so that persons or other products or systems have the degree of data access appropriate to their types and levels of authorization"* [10]. In ISO 25010, security is composed of five characteristics:
   - **Confidentiality:** ensures that data are accessible only to those authorized.
   - **Integrity:** prevents unauthorized access or modifications.
   - **Non-repudiation:** actions or events can be proven to have taken place.
   - **Accountability:** actions of an entity can be traced uniquely to the entity.
   - **Authenticity:** the identity of a subject or resource can be proved to be the one claimed.

   In Sigrid, security can be measured using multiple models:
   - ISO 5055 - Security
   - CWE Top 25 Most Dangerous Software Weaknesses (2023)
   - OWASP Low-Code/No-Code Top 10 (2022)
   - PCI DSS v4.0 - Control Objectives (2022)
   - SIG Security
   - OWASP Top 10 (2021)
   - OWASP ASVS 4.0 - Sections
   - OWASP ASVS 4.0 - Chapters

   Veer [21] describes the SIG Quality Model for Security, using the metrics shown in Figure 3.5. These

models and metrics aim to protect the system from malicious activities, ensure data safety, and address how securely the software handles data, manages user interactions, and maintains operational integrity. Measurements regarding security features like encryption strength, authentication mechanisms, and access control do not intuitively have an impact on the ease of refactoring a code base. While better security leads to fewer vulnerabilities and higher confidentiality, it typically focuses on adding or improving protective measures rather than fundamentally altering the structural code. Therefore, it does not significantly reduce the lines of code or the effort required for developers to refactor the software for sustainability. Greenability focuses on a developer's process when addressing sustainability aspects such as reducing energy consumption, which is outside the scope of security concerns. While both are important, they serve different purposes and involve different sets of criteria, resulting in security not being analysed further in this project.

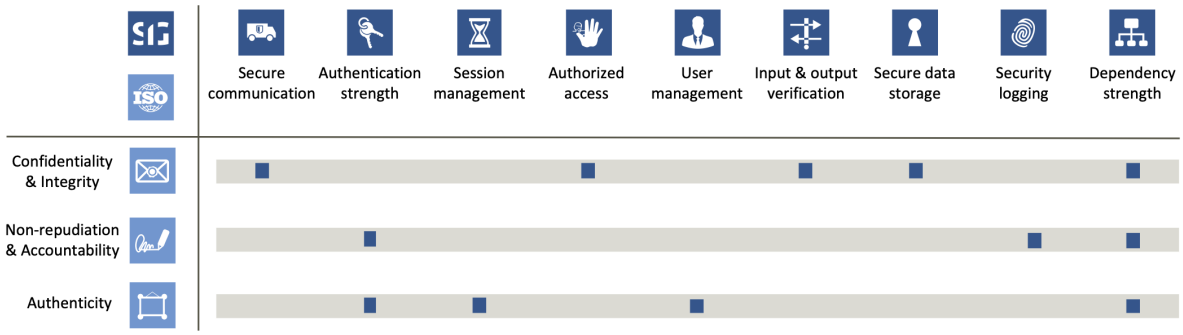


Figure 3.5: The mapping of system properties to the ISO 25010 security characteristics [21].

There is one exception when considering the metrics used in the SIG Security model, particularly measuring *Dependency Strength*. This metric ensures dependencies are managed, kept up to date ("fresh") and contain no known vulnerabilities [21]. This concept is very closely related to Open Source Health metrics (*Freshness, Vulnerability*, and *Management Risk*), which are already accounted for in Property Selection 6. For this reason, Security is not considered in the Greenability model for this project.
The tool used in the validation of this project, Sigrid, calculates Security based on the OWASP Top 10[2], rather than the SIG Security model. The OWASP Top 10 is a standard awareness document for developers and web application security, that represents a broad consensus about the most critical security risks to web applications. It is worth mentioning the model Sigrid uses, however considering both models measure the same concept, it does not impact the choice of excluding Security in Greenability.

5. *Open Source Health* is represented by a model by SIG that measures multiple metrics that indicate whether a software system is correctly managing its open-source third-party dependencies [45]. This model does not specify sub-characteristics like the other models do but is rather measured using 5 metrics (discussed in further detail in Section 3.2). Properly managing open source components through regular updates (freshness), active community support (activity risk), and efficient dependency management (management risk) ensures that the software remains maintainable and adaptable. This reduces the effort required for developers to implement Green IT patterns and refactor code efficiently. By maintaining a healthy open-source ecosystem, software projects can use the latest technology optimizations, leading to more sustainable and easily upgradable systems. This proactive management minimizes technical debt and enhances the software's ability to integrate green software patterns and refactorings with less effort, therefore improving Greenability. This property is therefore selected for the Greenability models as follows:

> **Property Selection 6:** Software with a higher Open Source Health score will require less effort to improve sustainability.

[2] `https://owasp.org/www-project-top-ten/`

6. *Cloud Readiness* is a model in Sigrid measured using the Twelve-Factor App Model [46]. Developed by Adam Wiggins, a co-founder of Heroku [47], The Twelve-Factor App describes many well-tested architectural patterns and best practices for software-as-a-service (SaaS) applications. This framework became influential in ensuring web apps are created with portability, resilience, and easy deployment and maintenance in modern cloud environments. These twelve best practices are as follows:

    - **Codebase:** Maintain a single codebase tracked in version control, with multiple deployments.
    - **Dependencies:** Explicitly declare and isolate dependencies to ensure the app is self-contained.
    - **Config:** Store configuration settings in the environment, separate from the codebase.
    - **Backing Services:** Treat backing services, such as databases and message queues, as attached resources.
    - Build, Release, Run: Strictly separate the build, release, and run stages of the deployment pipeline.
    - **Processes:** Execute the app as one or more stateless processes.
    - **Port Binding:** Expose services via port binding.
    - **Concurrency:** Scale out by running multiple instances of the app's processes.
    - **Disposability:** Maximize robustness with fast startup and graceful shutdown.
    - **Dev/Prod Parity:** Keep development, staging, and production environments as similar as possible.
    - **Logs:** Treat logs as event streams.
    - **Admin Processes:** Run administrative tasks as one-off processes

    The Cloud Readiness model on Sigrid does not have the functionality of reporting a final rating out of 5 stars for the external API for the system under consideration, unlike the other models used by this tool. This is a major disadvantage as aggregating final values for Greenability relies on each sub-property having a bench-marked rating. Benchmarking Cloud Readiness in Sigrid is an important feature that can be implemented in future work.

    Regardless of how the results are presented, this property is not used in the Greenability Model of this project. Cloud Readiness focuses on optimising applications for cloud deployment and operations, emphasising principles like stateless processes, dependency isolation, and environment configuration. While these metrics are important for efficient cloud performance, they do not directly relate to greenability. Greenability is concerned with reducing the effort required to implement sustainable practices across multiple software architectures and development environments, not just cloud-based systems.

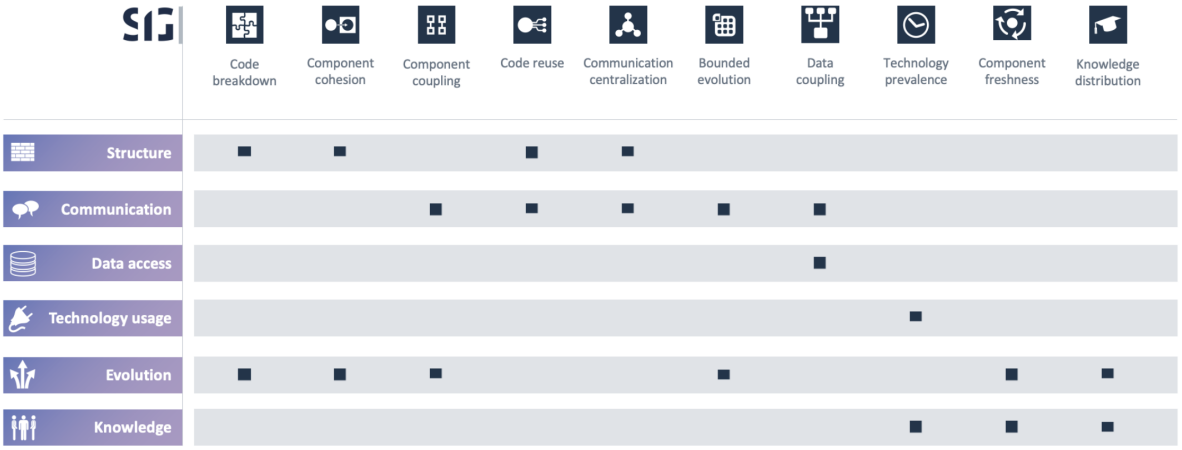


Figure 3.6: Architecture Quality [20]

7. *Software Architecture* is defined by Bijlsma and Olivari [20] as the degree to which an architecture is flexible and adaptable to change. It primarily measures metrics related to modularity, whether components can be worked on in isolation or whether new functionality can be added or removed with ease. Figure 3.6 displays the 6 sub-characteristics of Architecture Quality and their respective metrics. These sub-characteristics are defined as follows:

   - **Structure:** The arrangement of and relations between the parts or elements of an architecture.
   - **Communication:** The complexity of imparting or exchanging data throughout the architecture.
   - **Data Access:** Ease and efficiency of accessing or retrieving data stored within a database or other repository.
   - **Technology Use:** Degree to which technologies are common and standardized across the architecture.
   - **Evolution:** Degree to which changes can be made in isolation across the architecture.
   - **Knowledge:** Degree of technical knowledge distribution among team members within the organization.

   A well-structured architecture with modular components allows for easier maintenance, updates, and refactoring. This concept directly impacts the effort required to implement sustainable practices, making it applicable to Greenability. For example, changes to a modular component can be made without the risk of introducing errors or issues in other parts of the system. Effective communication within the architecture ensures that data flows seamlessly between components, making it easier to understand and refactor the system. This clarity also helps identify areas where green IT patterns can be applied. the more specific metrics used to measure these sub-characterisitcs are discussed in Section 3.2. Overall, well-designed software architecture improves Greenability, and it is therefore selected as a Greenability property:

   Property Selection 7: Software with a higher Architecture Quality score will require less effort to improve sustainability.

Source 3/3: Sustainable Software Quality Properties

The final source which provided valuable insight into common software quality properties is Albertao *et al.* [28, 29]. This paper introduces a set of software engineering properties designed to assess and improve the economic, social, and environmental sustainability of software projects. It is important to note the definition of *Sustainability* used in this paper refers to *"development which meets the needs of the present without compromising the ability of future generations to meet their own needs"*. Although they use the alternative definition of Sustainability, they still briefly describe the environmental impact of each property. A summary of the properties and sub-characteristics from this paper can be found in Table 3.2.

Category	**Property**	**Sub-characteristic**
Development	Modifiability + Reusability	Instability
		Abstractness
		Distance from main sequence
	Portability	Estimated System Lifetime
	Supportability	Support Rate
		Estimated Installation Time
Usage	Performance	Relative Response Time
	Dependability	Defect Density
		Testing efficiency
		Testing effectiveness
	Usability	Learnability
		Effectiveness
		Error Rate
		Satisfaction
	Accessibility	Support for users: motor impaired, visual-impaired, blind, illiterate, language and cognitive disabilities
		Internationalization support
		Localization support
Process	Predictability	Days project is delayed
		Estimation Quality Rate
	Efficiency	Project efficiency
	Project's Footprint	Work from home days
		Long-haul road trips

Table 3.2: Metrics associated with sustainability properties [28, 29].

Each of these properties is defined as follows. Once again, in the case where a property selection has already been made, it is referenced and not stated a second time.

- *Performance* is accounted for in Property Selection 4.
- *Portability* is accounted for in Property Selection 5.
- *Usability* as defined in Albertao *et al.* [28] in the same context as Heitlager *et al.* [27] and the ISO 9126 [13], and is therefore accounted for in Property Selection 3.
- *Efficiency* is defined by Albertao *et al.* [28] as being measured by *Project Efficiency*, rather than metrics regarding performance efficiency, for example execution time. They categorise *Performance* as a separate property, as stated above. *Efficiency* in this context is not applicable to Greenability.
- *Supportability, Accessability, Predictability* and *Project's Footprint* given their definitions, are not applicable to Greenability.
- *Dependability* is determined by three sub-characteristics: Defect Density, Testing Efficiency and Testing Effectiveness. The latter two metrics regarding testing could be applicable to Greenability, however their definitions are not. Albertao *et al.* define *Testing Efficiency* as `Defects found / Days of testing` and *Testing Effectiveness* as `Defects found and removed / Defects found`.

The concept of `hours of testing` cannot be accurately and realistically measured in most environments as there are too many confounding variables: the skill level of the developer, the complexity of the code being tested, the availability of testing resources, varying testing methodologies, and environmental factors such as interruptions or multitasking, all of which can significantly impact the actual time spent and the effectiveness of the testing process.
In addition to this, `Defects found and removed` requires a very specific tracking and documentation process to accurately determine the number of defects identified. This typically involves detailed logging of detected issues, tracking their status through a defect management system, and rigorous verification after fixes are applied. Unfortunately, with the current resources and setup of this project, considering only static source code analysis is used, such tracking is not feasible and is therefore not used in this project.

- Similarly to *Dependability*, the definitions used to determine *Modifiability* and *Reusability* are not ideal for this project. Firstly, *Abstractness* is language-specific and therefore not ideal for a model which is intended to be as generalised and framework-independent as possible. Secondly, the definition of *Distance from main sequence* and *Instability* measures the potential impact of changes in a given package as `I = Ce/(Ca + Ce)`, where Afferent Couplings (Ca) are *"The number of classes outside a package that depends upon classes within the package."* and Efferent Couplings (Ce) are *"The number of classes inside a package that depend upon classes outside the package."*. This value is essentially already represented by other selected metrics: the Maintainability model's *Module Coupling* is very similar to *Efferent Coupling* and *Afferent Coupling* is very similar to *External Coupling* in the Architecture Quality model. Therefore although it is applicable, it is not selected as a property for the Greenability Model.

Summary of Greenability Properties and Sub-Characteristics

From analysing these three sources, seven property selections are made. These properties and their sub-characteristics are summarised in Figure 3.7.

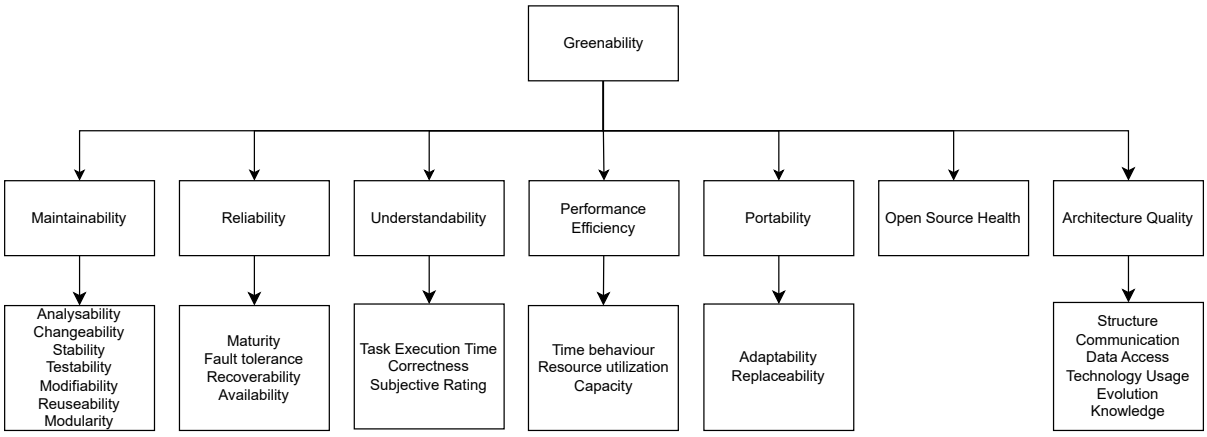


Figure 3.7: Greenability Properties and Sub-Characteristics.

3.2 Analysis of Greenability Metrics

This step determines which metrics need to be measured to calculate each Greenability property. The properties selected for Greenability are Maintainability, Reliability, Understandability, Performance Efficiency, Portability, Open Source Health, and Architecture Quality. The metrics for each property are as follows:

1. *Maintainability:* According to Heitlager *et al.* [27] and Vis *et al.* [19], the metrics measured by Sigrid that contribute to maintainability, and their applicability to Greenability, are as follows:

    - **Volume:** The overall size of the source code of the software product. Size is determined from the number of lines of code per programming language normalized with industry-average productivity factors for each programming language. Higher volume decreases analyzability as it takes more time and effort to understand, edit, and refactor large code bases. This would decrease Greenability because greater effort is required for sustainable refactoring.
    - **Duplication:** The degree of duplication in the source code of the software product. Duplication concerns the occurrence of identical fragments of source code in more than one place in the product. If a component can be refactored to become more green, each duplication of that component requires additional effort to change. This results in duplication decreasing changeability and analyzability, and hence decreasing Greenability.
    - **Unit Size:** The size of the smallest executable parts of the source code, such as methods or functions, in terms of the number of source code lines. Larger units decrease analyzability and testability, as they are harder to understand and test comprehensively. Therefore, a large unit size decreases Greenability.
    - **Unit Complexity:** The degree of complexity in the units of the source code. Complex units are more difficult to refactor and test, which decreases changeability and testability, and hence Greenability.
    - **Unit Interfacing:** The number of interface parameter declarations in the units of the source code. Higher unit interfacing increases execution time (performance) and decreases changeability as each caller has to be refactored when a parameter changes. Therefore, higher unit interfacing decreases Greenability.
    - **Module Coupling:** The coupling between modules is measured by the number of incoming dependencies for the modules. A higher number of incoming dependencies results in higher coupling, which decreases modifiability and hence decreases Greenability.
    - **Component Independence:** The percentage of code in modules that have no incoming dependencies from modules in other top-level components. Separating components into an interface that receives incoming communication from other components ensures that changes to the interface do not affect internal methods. Having a smaller interface with fewer dependencies connected to other components increases changeability and hence Greenability.
    - **Component Entanglement:** The percentage of communication between top-level components that are part of commonly recognized architecture anti-patterns. Minimizing component entanglement increases maintainability, as fewer dependencies increase changeability and hence increase Greenability.
    - **Test Code Ratio:** The percentage of the ratio of "test code lines" to "production code lines". Note that this number is not the same as "test coverage", which is measured dynamically. 100% test code ratio roughly translates to having 80% test coverage [17]. These values are empirical benchmarks that provide a balance between practicality (statically measured) and code correctness assurance. A high test code ratio implies a high test coverage, which ensures code functionality remains unchanged after green refactorings are implemented, as well as ensures a low risk of introducing new bugs to the system. This provides a safety net for refactorings, improving the ease of making software more green, hence improving greenability.

    Metric Selection 1: Maintainability can be measured using volume, duplication, unit size, unit complexity, unit interfacing, module coupling, component independence, component entanglement and test code ratio.

2. *Reliability:* Reliability is the degree to which a software system consistently performs its intended functions without failure, ensuring stability, and robustness across various operating conditions [13]. Sigrid measures reliability using 10 metrics, which each correspond to several CWEs.

Please refer to Section 2.3 for a description of CWEs and the ISO 5055.
It is not feasible to list all CWEs used in this model, for example, *Logic and Data Flow* identifies 50 CWEs. A shorter example that can be shown is for the first metric *Inter-process communication*, which identifies the following four CWEs:

- CWE-111: Direct Use of Unsafe JNI
- CWE-770: Allocation of Resources Without Limits or Throttling
- CWE-774: Allocation of File Descriptors or Handles Without Limits or Throttling
- CWE-1088: Synchronous Access of Remote Resource without Timeout

Rather than listing the specific weaknesses identified for each metric, a description of the general errors applicable to each are given, along with their relevance to Greenability. These are as follows:

- **Inter-process Communication:** This metric identifies weaknesses with respect to design patterns (eg. circuit breaker, timeouts, retries). Some of these patterns are also mentioned by the GSF [3], indicating a consensus in the community about the relevance of these patterns to Green IT. Efficient inter-process communication ensures smoother coordination between different software components, making it easier to modify or extend the system in the future. Better inter-process communication improves Greenability.
- **Logic & Data Flow:** This metrics measures the correctness of if statements. For example contradicting if statements or incorrect boolean logic results in unreachable code, or stating a variable should be both larger and smaller than zero can never be true. This leads to bugs in logic and data. Better logic and data flow reduce the occurrences of these bugs and hence reduce complications when changes are made implementing green software patterns. Better logic and data flow also make it easier for developers to understand how data is processed and manipulated within the system, improving Greenability.
- **Memory Allocation & Release:** This metric identifies bugs involving not correctly freeing up memory. Good memory management improves reliability by reducing risks of memory leaks, performance degradation, and memory-related issues in testing. Therefore good memory allocation and release reduces potential bugs, making implementing green software practices easier, resulting in higher greenability.
- **Concurrency & Synchronization:** This metric identifies bugs related to deadlocks, locking issues, or not synchronizing code being used in a multi-threaded environment. Well-designed concurrency and synchronization techniques minimize the need for extensive code changes when adding or modifying concurrent features. It also allows the software to effectively make use of parallelism and resource utilization and helps developers monitor and analyse system behaviour. This in turn increases greenability.
- **Pointers:** This metric analyses the handling of pointers and recognizes bugs such as null pointer exceptions. The proper handling of pointers improves software reliability by reducing the risk of memory corruption or security vulnerabilities. This minimizes the likelihood of pointer-related errors or bugs that could impact sustainability refactorings, hence improving greenability.
- **Resource Management:** Efficient resource management practices improve how the software handles varying workloads and resource constraints, as well as reduce the overhead associated with refactoring efforts by ensuring that system resources are utilised efficiently, minimizing the need for extensive modifications. This makes refactoring easier and improves greenability.
- **Hard-coded Configuration:** This metric identifies hard-coded features, for example, URLs, as having a negative impact on reliability, because if a change needs to be implemented the system will have to be redeployed. Flexible configuration options make it easier to adjust system behaviour or settings without modifying the code base, as well as reduce the effort required to implement or test changes related to energy-saving features or optimizations. Hard-coded configuration settings are less flexible, harder to change, and decrease greenability.
- **Error Handling:** This metric identifies bugs such as the wrong usage of try-catch mechanisms. Good error-handling mechanisms improve the software's reliability by preventing system crashes or unexpected failures. Improved detection and diagnosis of errors or exceptions improves greenability.
- **Arithmetic Operations:** This metric identifies bugs such as integer overflow, division by zero, forgetting precedence of operations and incorrect parenthesis. Efficient arithmetic operations min-

[3] `https://greensoftware.foundation/`

imize computational overhead and improve the software's performance, hence improving greenness. Not only will fewer bugs improve the ability to implement green software practices, but reducing the complexity of mathematical algorithms will make it easier for developers to understand and modify the code, hence improving greenability.

- **Dead, Irrelevant & Obsolete Code:** This metric identifies code that is never executed or commented out code and is measured on the method level. Removing dead, irrelevant, or obsolete code improves software's maintainability and understandability by reducing technical debt and making it easier to identify hot spots for energy-saving optimizations, hence improving greenability.

As seen in the explanations above, all Reliability metrics are relevant to Greenability. Because this model is a findings-based model and not a metrics-based model, the star rating is calculated for Reliability as a whole, rather than each metric having a 5-star rating. Reliability is therefore selected as a Greenability property as follows:

> **Metric Selection 2:** All metrics representing Reliability can be summarised in one value representing Reliability.

3. *Understandability:* The metrics Lavazza *et al.* [15] utilised to measure understandability apply to Greenability as they can be measured using static analysis methods and source code analysis. These metrics include:

   - **Logical Lines of Code (LLOC):** This metric measures the same concept as the volume metric used in the maintainability property. Fewer lines of code reduce the number of instances where green refactorings can take place, therefore making it easier to implement green practices, and improving greenability.
   - **McCabe's Complexity (McCC):** Although originally used to measure maintainability and testability, the control flow of units can also be indicative of understandability. This metric is also accounted for within maintainability. Lower McCC indicates simpler control structures, which facilitate easier refactoring for sustainability by reducing the cognitive load on developers.
   - **Nesting Level Else-If (NLE):** This metric measures the depth of nesting in control structures. Lower nesting levels make code easier to understand and refactor, reducing the effort needed to implement sustainable practices, hence improving greenability.
   - **Halstead Volume (HVOL) [48]:** This metric measures program length and vocabulary using number of occurrences of operators and operands. Lower HVOL indicates simpler code, making it easier to understand and refactor for sustainability, thus reducing the effort required for green practices. HVOL is calculated as:

   $$HVOL = N * \log_2(\eta) \tag{3.1}$$

   where $N = N_1 + N_2$ is the "program length," N_1 is the total number of occurrences of operators, N2 is total number of occurrences of operands; $\eta = \eta_1 + \eta_2$ is the "program vocabulary," η_1 is the number of distinct operators and η_2 is the number of distinct operands.
   - **Halstead Calculated Program Length (HCPL):** This metric also considers operators and operands in calculations similarly to HVOL. HCPL is calculated as:

   $$HCPL = \eta_1 * \log_2(\eta_1) + \eta_2 * \log_2(\eta_2) \tag{3.2}$$

   - **Maintainability Index** [49] uses a combination of HVOL McCC and LLOC:

   $$MI = 171 - 5.2 * \ln(HVOL) - 0.23 * (McCC) - 16.2 * \ln(LLOC) \tag{3.3}$$

   - **Cognitive Complexity (CoCo)** was introduced by SonarSource [50] as a variant of McCabe's complexity, as it assigns a weight equal to a decision point's nesting level plus 1. For example, the following code example:

```
void someMethod() {
  if (condition1)
    for (int i = 0; i < 100; i++)
      while (condition2) { ... }
}
```

the `if` statement at nesting level 0 has weight 1, the `for` statement at nesting level 1 has weight 2, and the `while` statement at nesting level 2 has weight 3. This results in this code having $CoCo = 1 + 2 + 3 = 6$, compared to $McCC = 4$.

Metric Selection 3: Understandability can be measured using Logical Lines of Code, McCabe's Complexity, Nesting Level Else-If, Halstead Volume, Halstead Calculated Program Length, Maintainability Index and Cognitive Complexity.

4. *Performance Efficiency:* The ISO 25010 [10] defines Performance Efficiency as having three sub-characteristics:
   - **Time behaviour:** *"The extent to which the system's response time and throughput meet requirements"*
   - **Capacity:** *"Maximum growth possible within time behavior requirements"*
   - **Resource utilization:** *"Computing resources required to meet time behavior and capacity requirements"*

   Sigrid measures Performance using CWEs, following the ISO 5055 [14]. The five categories which identify CWEs are:
   - Inefficient Resource Usage
   - Inefficient Computation
   - Expensive Operations in Loops
   - Inefficient Data Query
   - Inefficient Casting or Boxing

   Each of these identifies various CWEs. Similarly to Reliability, not all of these are stated in this report, however, an example is *Inefficient Computation* which identifies the following:
   - CWE-407: Inefficient Algorithmic Complexity
   - CWE-1176: Inefficient CPU Computation
   - CWE-1333: Inefficient Regular Expression Complexity

   It is clear from current literature, for example, Sahin *et al.* [8], that software performance efficiency and energy consumption are strongly correlated: if a program executes for longer, it uses more electricity. This is however not the only way Performance Efficiency can contribute to sustainable software. Firstly, efficient systems make it easier to pinpoint areas where energy consumption can be reduced. By collecting detailed metrics developers can analyse patterns and identify components that consume high amounts of resources or energy. This makes it easier to target refactorings aimed at improving energy efficiency. Secondly, for software to improve in energy consumption, its performance must be observable and measurable. By measuring and monitoring performance efficiency, developers can compare different versions of the software to see the impact of green refactorings. This ability to observe and measure performance metrics is needed to show the effectiveness of green software patterns.

   Although these measurements seem to meet the criteria of this project, they contain a critical limitation: Sigrid does not report a Performance score on a scale from 1 to 5 as it does for other metrics [4]. The model used in Sigrid instead reports on several findings on a severity scale from low to critical. Due to this limitation, this property cannot be validated in this report. The validation process is described in Section 4.4. Since this metric cannot be measured for this project, exploring dynamic methods and metrics for evaluating performance can provide an interesting alternative approach. This information is presented in the Related Work Section 7.2.

   Although is it not possible to validate this selection in this project, it is still included in the final model and noted as an area for future work.

   Metric Selection 4: Software Performance Efficiency can be measured statically by identifying CWEs that indicate Inefficient Resource Usage, Inefficient Computation, Expensive Operations in Loops, Inefficient Data Query and Inefficient Casting or Boxing.

[4] This is a feature which could be implemented in the future through bench-marking Performance Efficiency similarly to the other models used by this tool.

5. *Portability:* is the ability to move software among different runtime environments without having to rewrite it partly or fully, and contains three sub-characteristics according to ISO 25010 [10]: installability, adaptability and replaceability. The second two have been selected for the Greenability model, as described in Section 3.1. Lenhard and Wirtz [32] and Lenhard [31] discuss metrics for each of these three sub-characteristics, along with an additional set of metrics to measure *direct portability*.

   - **Installability:** Installability metrics include Ease of setup retry, Installation effort, Deployment Flexibility and Deployment Effort, however as stated previously, these are not considered for Greenability, as only Adaptability and Replaceability are applicable.
   - **Adaptability:** The adaptability of a process element is quantified by counting the number of alternative representations for the functionality provided by the element that results in the same runtime behaviour. These alternatives represent different ways the element can be implemented within a process language.

     $$AM(p) = \frac{1}{n} \sum_{i=1}^{n} \frac{AS(e_i)}{R} \tag{3.4}$$

     where $AM(p)$ is the adaptability metric for process p, $AS(e_i)$ is the adaptability score for element e_i (the cardinality of the set of these alternatives), and R is the reference value. The metric provides a percentage value ranging from 0 to 1, making it easy to understand and interpret.
   - **Replaceability:** Lenhard [31] explain that following the ISO definition, replaceability cannot be evaluated for a single isolated piece of software, but requires a paired combination of two pieces: the currently installed software and a candidate for replacement. They also explain the link with *similarity*, in that *"processes are more likely to replace each other if they are highly similar to each other"*. Similarity can be measured based on dependency graphs, causal behavioural profiles, causal footprints, and the string edit distance of sets of traces. Although no dedicated metrics have been proposed, it is clear Replaceability metrics are computed on an ad hoc basis to find an alternative process, and are not suitable for continuous inspection.
   - **Direct Portability:** These equations are extensive and take multiple considerations into account, therefore only the basic equation is shown. For more information please refer to the paper.

     – **Basic Portability Metric (M_b):** This metric quantifies the degree of portability for a process by considering the complexity of modifying the process for a new platform compared to rewriting it from scratch. It uses the number of process elements that need to be rewritten when porting. It relates a number of problematic elements to the total number of elements.

     $$M_{port}(p) = 1 - \frac{C_{port}(p)}{C_{new}(p)} \tag{3.5}$$

     where $M_{port}(p)$ is a metric that quantifies the degree of portability for a process definition p. $C_{port}(p)$ is the cost of modifying the process definition so that it can run on another platform (LOC that has to be rewritten when porting it). $C_{new}(p)$ is the cost of rewriting it completely for a new platform (total LOC).

     – **Weighted Elements Portability Metric (M_e):** This metric improves on the basic portability metric by taking into account the degree of severity of each element's portability issues. It calculates the complexity of porting each element based on how many engines support it.

     – **Activity (M_a):** This metric focuses on the activities within a process, considering only the control-flow-related aspects such as activities, gateways, and events. It measures the impact of problematic activities on portability. It is similar to M_e, but is limited to activities instead of elements.

     – **Service Communication (M_s):** This metric considers only the activities related to message sending and reception, focusing on the critical aspects of communication relationships that affect process portability. It is similar to M_e, but is limited to activities for service communication instead of elements.

Metric Selection 5: Metrics used to measure Portability include the Adaptability Metric, Basic Portability, Weighted Elements Portability, Activity Portability and Service Communication Portability.

6. *Open Source Health:* The model used by Sigrid [17] measures multiple metrics for Open Source Health that indicate whether a software system is correctly managing its open source third-party dependencies. All quotations mentioned are taken verbatim from Sigrid documentation [45]. These metrics are as follows:

    - **Vulnerability Risk:** It is recommended to *"avoid using vulnerable dependencies as vulnerabilities can lead to severe consequences, such as data breaches, or compromised system functionality."* While addressing vulnerabilities is crucial for security, it does not significantly impact the effort required for implementing sustainable software practices. Vulnerability fixes are more about security than sustainability and are therefore not applicable to Greenability.
    - **Legal Risk:** It is recommended to *"avoid dependencies with restrictive licenses. With incompatible or restrictive licenses, software products might expose themselves to legal risks. Such restrictive licenses may conflict with the end-user, with the distribution method, or the business model, and failure to comply with licensing terms can lead to legal repercussions."* This is not applicable to Greenability as legal risks pertain to compliance and licensing issues, which do not directly affect the structural code or the effort needed to implement sustainability practices.
    - **Freshness Risk:** It is recommended to *"keep dependencies up-to-date. Maintaining third-party dependencies up-to-date is crucial for both product security and future-proofing. Outdated dependencies not only pose security threats but also increase maintenance efforts, as updating minor releases is less resource-intensive than major releases."* This is applicable to Greenability for multiple reasons. Fresher dependencies typically require less effort and resources to update and maintain compared to older or outdated dependencies. Keeping dependencies fresh often includes more efficient and updated technologies, which can reduce the effort required for refactoring code for sustainability. Newer dependencies can also be easier to maintain and optimize for sustainability. Minor release updates often involve incremental changes and bug fixes, which are generally less disruptive and resource-intensive than major updates that introduce significant changes or require extensive refactoring. For all of these reasons, fresher dependencies result in higher greenability.
    - **Management Risk:** It is recommended to *"manage dependencies with a package manager, essential for efficient software development, central version setup, simplified dependency resolution, and reduced compatibility risks."* This could be seen to be partially applicable to Greenability as proper management of dependencies can simplify the process of maintaining and updating the software, indirectly supporting sustainability by reducing the complexity and effort involved in implementing green refactorings.
    - **Activity Risk:** It is recommended to *"use dependencies with active communities. An active support community significantly influences the reliability and effectiveness of e.g., security patches, and timely issue resolutions should it be needed."* Dependencies with active communities ensure ongoing support and updates, which can reduce the effort needed to refactor and maintain the software for sustainability. Active support can lead to quicker adoption of sustainable practices and hence is applicable to Greenability.

    Metric Selection 6: Open Source Health metrics applicable to Greenability are Freshness Risk, Management Risk and Activity Risk.

7. *Architecture Quality:* Bijlsma and Olivari [20] define Architecture quality as consisting of 6 sub-characteristics (explained in Section 3.1) and 10 metrics. Each of these 10 metrics is considered as metrics for Greenabilty, with their explanations for their applicability as follows:

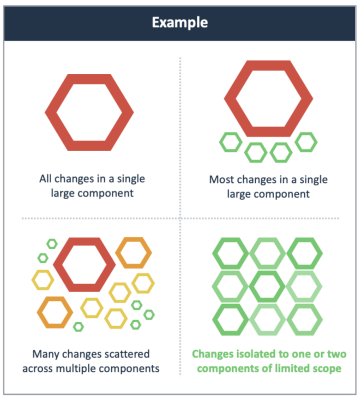
    - **Code breakdown:** Measures the level of modularization in the code base. Figure 3.8a shows this visually. A highly broken-down code base has small, independent components that are easier to understand, maintain, and replace. High modularization makes it easier to isolate and implement green refactorings, reducing the effort required to adopt sustainable practices. Therefore broken down code improves greenability.
    - **Component coupling:** Measures the degree to which components depend on other components. Highly coupled components are more difficult to maintain and evolve independently. Lower coupling reduces the complexity of making sustainable changes, allowing for easier refactoring and implementation of green practices, hence improving greenability.
    - **Technology Prevalence:** Measures the degree to which a system is built in modern and common technologies. Modern technologies are more sustainably conscious, more likely to have tools, li-

braries, and frameworks that support sustainable development practices, and more likely to be designed to take advantage of the latest hardware advancements which utilise hardware capabilities effectively. The higher the degree to which a system is built in modern and common technologies, the higher the greenability.

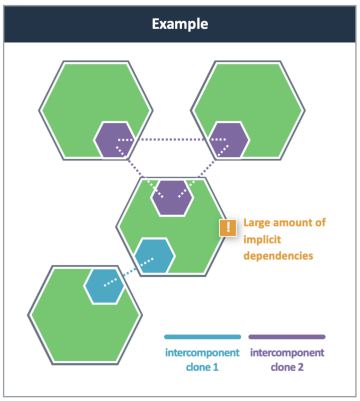
- **Component cohesion:** Measures the degree to which components encapsulate specific business responsibilities. High cohesion means that components are well-defined and encapsulated, making them easier to understand, modify, and refactor. When components are cohesive with few dependencies, developers can make changes within a component without affecting other parts of the system. This also makes it easier to identify areas that impact energy efficiency. Therefore high component cohesion improves Greenability.
- **Code Reuse:** Measures the extent of code duplication across components. Figure 3.8b shows this visually. High code reuse means less duplication, leading to easier maintenance and updates. This concept differs slightly from the *Duplication* metric measured in the Maintainability model as it is not calculated as the number of duplicated LOC, but rather *"Lines of code for files in the component, where those files are not located within one of the component's sub-components."*. Lower code reuse improves Greenability.
- **Communication centralization:** Measures how centralized communication is within components. It is measured as the percentage of code within a component that is not involved in direct communication with other components. Figure 3.8c shows this visually. Centralized communication reduces dependencies and increases encapsulation. *"When calls from a component to other components are not centralized, encapsulation is low, resulting in a component that is increasingly more sensitive to changes in the "outside world" and makes it more difficult to update the component if such changes occur."* Centralized communication hence reduces the complexity of implementing sustainable changes, making green refactoring easier, improving Greenability.
- **Data coupling:** Measures the degree to which components depend on shared data stores. Figure 3.8d shows this visually. High data coupling makes it difficult to change data structures without affecting multiple components. Lower data coupling facilitates isolated changes to data structures, reducing the effort needed to implement sustainable data management practices. This is very closely related to the *Module Coupling* metric measured in the Maintainability model. The notion of module in this case corresponds to a grouping of related units. This distinction between data stores and modules justifies the applicability of both metrics with Greenability.
- **Bounded evolution:** Measures the degree of co-evolution of components within a system based on the frequency of coupled code modifications over time. Co-evolving components are defined as components modified together at a regular frequency and indicate implicit functional relationships within a code base. Figure 3.8e shows this visually. It is measured as a percentage of changes made to the component involved in co-evolution with other components. Reducing co-evolution allows for isolated changes, making it easier to implement green refactorings without affecting unrelated parts of the system, improving Greenability.
- **Knowledge distribution:** Measures the degree to which development can grow and retain knowledge over a given system. A well-distributed knowledge base reduces risks associated with key personnel leaving. Greenability is not inherently impacted by how well knowledge is distributed among team members. Knowledge distribution does not directly impact the structural code or the effort required for implementing sustainable practices and is therefore not applicable to Greenability.
- **Component freshness:** Measures the degree to which components are actively kept up to date and maintained. Figure 3.8f shows this visually. Fresh components are easier to maintain and modify. Regularly maintained components are more adaptable to sustainable practices, reducing the effort needed for green refactorings and keeping the system efficient and up-to-date. This concept aligns closely with Technology Prevalence and Freshness Risk (from the Open Source Health model). These are all applicable to Greenability.

Therefore all metrics except Knowledge Distribution are applicable to Greenability.

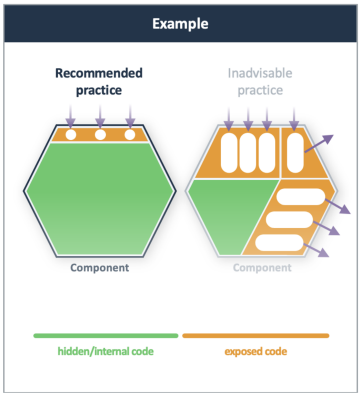
Metric Selection 7: Architecture metrics applicable to Greenability are Code Breakdown, Component Coupling, Technology Prevalence, Component cohesion, Code reuse, Communication centralization, Data coupling, Bounded evolution and Component Freshness.



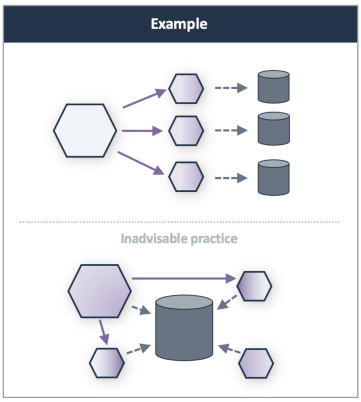
(a) Code Breakdown [20]



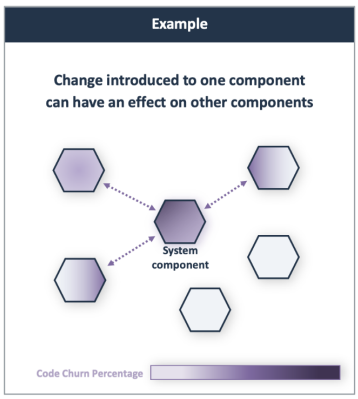
(b) Code Reuse [20]



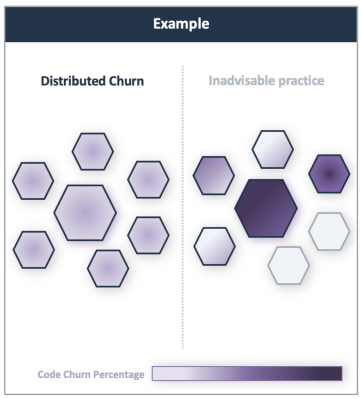
(c) Communication centralization indicates that the less internal code that interfaces with other components in the system, the better encapsulated a component is. [20]



(d) Data Coupling [20]



(e) Bounded Evolution [20]



(f) The Component Freshness metric indicates regular maintenance should be uniform to a system and distributed over multiple components, not centralized within a single component. [20]

Figure 3.8: Overview of Architecture Quality Metrics

3.3 Greenability Model Summary

The selection of properties, sub-characteristics and metrics as a result of the literature review is summarised in Figure 3.9.

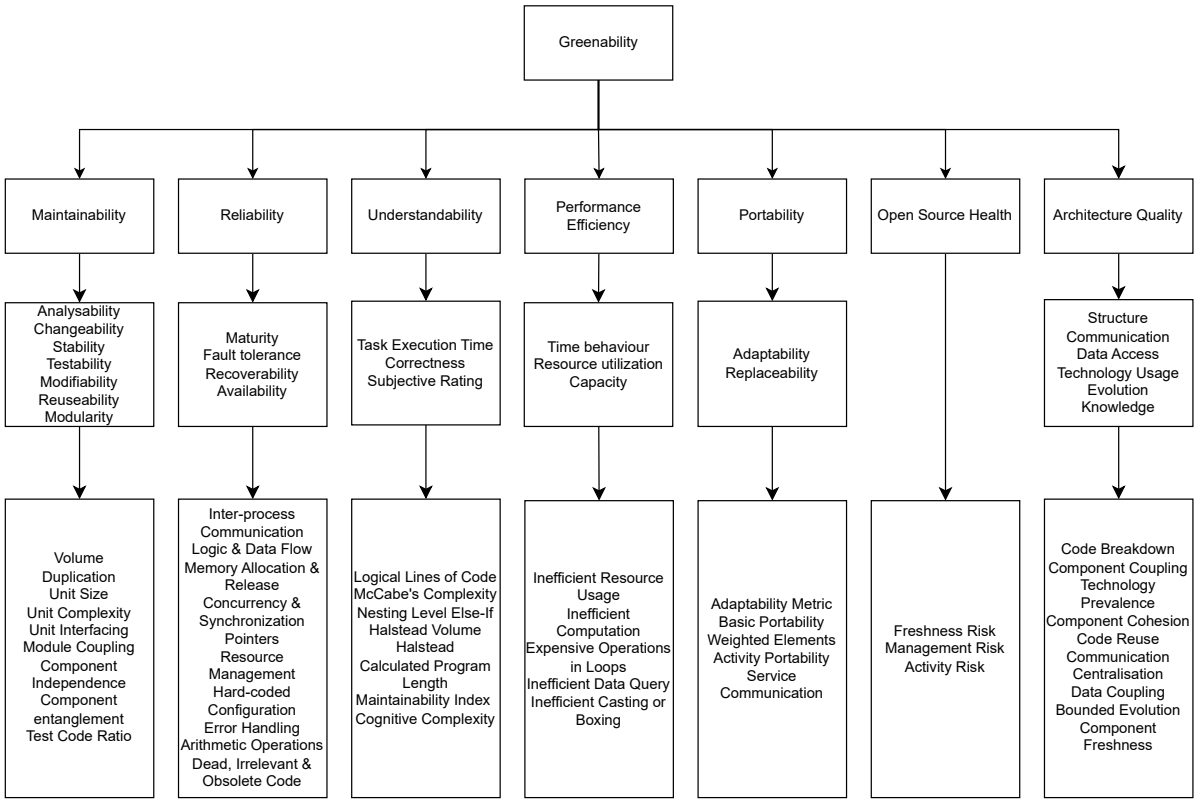


Figure 3.9: Greenability Properties, Sub-Characteristics and Metrics

Chapter 4

Validating the Software Greenability Model

In order to validate the selection of Greenability properties, sub-characteristics and metrics presented in Chapter 3, two validations are conducted: a correlation analysis and a survey.

The correlation analysis is performed by measuring Greenability metrics of 9 open source software systems and measuring the effort taken to implement 6 different code refactorings on each system. For this step, effort is defined as the number of lines of code added or changed. The goal of this validation is to support the selection of Greenability properties and metrics, and determine which metrics have the biggest impact on making the process of green software refactorings easier. This validation proves the selection of Greenability properties and metrics does in fact impact refactoring effort. A visualisation of this methodology can be seen in Figure 4.1.

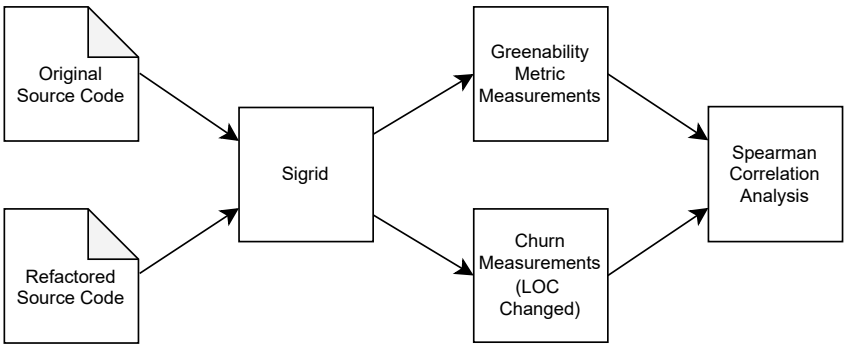


Figure 4.1: Correlation Analysis Methodology for Greenability Model Validation

The survey validation is performed with software engineers, developers and consultants. This survey presents participants with a subset of Greenability metrics, and gathers information on their perception of effort required to implement the same 6 code refactorings used in the correlation test, given various scores of the Greenability metrics. This aims to provide an additional perspective of refactoring effort by comparing the number of lines of code added or changed with developer's perceived effort, as well as how this effort changes depending on Greenabilty metric scores. A visualisation of this methodology can be seen in Figure 4.2.

This chapter explains the methodology supporting these two validation approaches. This process begins with finding a repository suitable for such experiments. Section 4.1 explains how the selected repository is found and why it is selected. Section 4.2 explains how the repository was created and what each of the 9 systems and 6 refactorings entails. Section 4.3 describes how these 9 systems are uploaded onto Sigrid (the static source code analysis tool) and how the results for both Greenability Metrics and churn results are retrieved from Sigrid. Section 4.4 describes how the correlational tests are performed on the results from Sigrid. Finally Section 4.5 explains how the validation survey is designed and how information is presented to the participants.

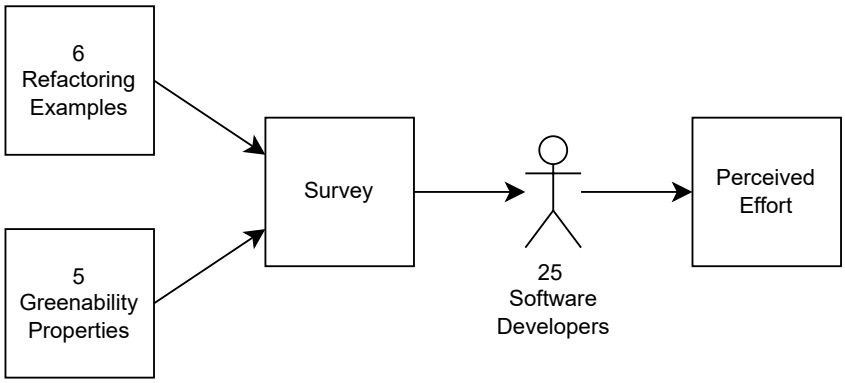


Figure 4.2: Survey Methodology for Greenability Model Validation

4.1 Repository Selection

The first task was to find an online repository containing before and after versions of systems where green software refactorings had taken place. The criteria being considered when searching for such repositories were as follows:

- Multiple systems of varying sizes and characteristics are refactored. This is to show the effect of Greenability metrics on churn results.
- Before and After versions of refactored systems are available, preferably through an online repository. This is required for churn results to be calculated.
- All refactorings were implemented on all systems. This is required for valid conclusions can be drawn from results. It will not be possible to compare two Greenability metrics from systems refactored in different ways.
- Valid refactorings were implemented. It is preferred for the refactorings to be specifically green software patterns that are proven to reduce energy consumption.

Table 4.1 shows all repositories considered, along with the papers associated with them.

Paper	Number of Systems	Number of Refactorings	All Refactorings implemented on all Systems	Before and After Versions Available	Refactoring Type
Sahin *et al.* [8]	9	6	✓	✓	Most Common
Cruz and Abreu [51]	6	6		✓	Performance Based
Longo *et al.* [6]	5	8		✓	Green
Vartziotis *et al.* [52]	6	3	✓	✓	AI
Cruz *et al.* [53]	150	4			Green
Hindle [54]	500	0		✓	Version history
Oliveira [55]	17	3	✓		Green
Palomba *et al.* [56]	60	9	✓	*	Code Smells
Ournani *et al.* [57]	11	5	✓		IO methods
Ournani *et al.* [58]	7	25		✓	Common Refactorings

Table 4.1: Repositories considered for Greenability Validation

* Palomba *et al.* [56] states *"The complete list of the apps used in this study is available in our online appendix:* `https://dx.doi.org/10.6084/m9.gshare.3759489`". This URL returns a not found error.

It had proven extremely challenging to find a repository which not only had before and after versions available, but also all refactorings implemented over all systems. Only one repository could be found that fit both those criteria: Sahin *et al.* [8]. This repository however still has two disadvantages:

1. **Data points:** The study refactored 9 systems each with 6 refactorings, and implemented the refactorings 4 times in each system. This resulted in $9 * 6 * 4 = 216$[1] files. Although this number seems high and statistically significant, because the 4 refactoring implementations per system were averaged for this project, there were only $9 * 6 = 54$ files used in this validation. It would be preferred to have a higher number of systems and refactorings.
2. **Impact of the refactorings on energy consumption:** The conclusion of this study was the refactorings chosen both *increase* and *decrease* energy consumption. It would be preferred for the refactorings being implemented to be specifically proven to *strictly reduce* energy consumption.

Despite these two disadvantages, this is the repository selected for the validation of this project. This repository was the best selection out of the options available, and the areas for improvement have been noted in the Threats to Validity Section 6.6.

For clarification on how this dataset was created and which refactorings were implemented, the paper by Sahin *et al.* [8] is explained in the next section.

4.2 Repository Explanation

"How Do Code Refactorings Affect Energy Usage?" by Sahin *et al.* [8]

Sahin *et al.* [8] is an empirical study exploring the impact of code refactorings on energy consumption. It investigates the energy effects on 9 software systems of 6 commonly-used refactorings, revealing that *"refactorings can not only impact energy usage but can also increase and decrease the amount of energy used by an application."*. The results conclude that integrated development environments (IDEs) could incorporate energy efficiency information to aid developers.

The refactorings implemented in this paper were selected according to the following criteria:

- **Common Usage:** The refactorings chosen should be commonly used by developers. This was determined by examining data from the Eclipse Usage Data Collector (UDC), which provided information on the most frequently used editing commands by developers.
- **Structural Changes:** The selected refactorings should make some structural changes to the application. This means that the changes should be reflected in the application's compiled bytecode. Refactorings that do not alter the structure, such as renaming variables or methods, were excluded.
- **Applicability:** The refactorings should be applicable to the Java applications used in the study. This was ensured by manually examining the code of each application to identify locations where the preconditions for applying each refactoring were met.
- **Multiple Data Points:** The refactorings should provide multiple data points for analysis. Refactorings that could only be applied in very specific circumstances and thus offer limited data points were not included.

The six refactorings selected were as follows:

- **Convert Local Variable to Field:** Creates a new field by turning a local variable into a field.
- **Extract Local Variable:** Creates a new variable assigned to the expression currently selected and replaces the selection with a reference to the new variable.
- **Extract Method:** Creates a new method containing the currently selected statement or expression and replaces the selection with a reference to the new method.
- **Introduce Indirection:** Creates a static method that can be used to indirectly delegate to the selected method.
- **Inline Method:** Copies the body of a callee method into the body of a caller method
- **Introduce Parameter Object:** Replaces a set of parameters with a new class, and updates all callers of the method to pass an instance of the new class as the value to the introduced parameter.

Examples of these refactorings, showing code snippets of before and after implementation in both a simple scenario and a real-life scenario, can be found in the Appendix A.

Eclipse's refactoring support was used to implement the refactorings. When applying the refactorings,

[1] This value was in practice 197 considering one refactoring could only be implemented 17 times instead of 36.

Eclipse provides configuration options for all considered refactorings. The options and the parameter values for those options for each refactoring are listed below. Please note these are quoted directly from the paper:

- **Convert Local Variable to Field:** The new field created by the refactoring can be made "public", "protected", or "private". They chose to make it public. Also, they chose to initialize the new field at its declaration location instead of in the current method, when it was possible.
- **Extract Local Variable:** All occurrences of the selected expression can be replaced by a reference to the newly created variable, or only the selected expression can be replaced. They chose to replace all occurrences.
- **Extract Method:** The extracted method can be created with "public", "protected", or "private" protection. They chose to make the extracted method public.
- **Introduce Indirection:** Either all method invocations can be redirected to the newly created static method, or only the selected method invocation can be redirected. They chose to redirect all method invocations.
- **Inline Method:** The method to be inlined can be inlined into every caller method or only into the selected caller method. They chose to inline it into every caller method if it is applicable.
- **Introduce Parameter Object:** The new parameter object class can be a top-level class or nested within the current class. They chose to create the new class at the top level. In addition, the signature of the existing method can be changed, or it can be modified to be a proxy method (i.e. the method simply packages its arguments in an instance of the new parameter object class and passes along the new object.) They chose to modify the method rather than keep it as a delegate.

The 9 systems used in this study, along with their characteristics, can be seen in Table 4.2. It is clear there is a wide range among these systems, which aligns well with representing the full scope of Greenability metrics.

Table 4.2: Java applications from Sahin *et al.* [8]

Name	Version	# Classes	# Methods	LoC	# Tests	% Coverage
Commons Beanutils	1.8.3	118	1,199	31,538	1,514	63
Commons CLI	1.2	21	192	4,739	187	96
Commons Collections	3.2.1	412	3,796	63,852	39,143	81
Commons IO	2.4	108	1,069	25,663	966	89
Commons Lang	3.1	147	2,219	55,626	2,047	94
Commons Math	3.0	666	4,974	135,796	3,451	83
Joda-Convert	1.2	10	65	1,317	105	93
Joda-Time	2.1	226	3,731	67,590	11,663	88
Sudoku	—	4	57	497	25	81

The 9 Java applications were selected based on several criteria to ensure a diverse and comprehensive analysis. An explanation of how these systems were chosen is as follows:

- **Variety of Application Domains:** The applications were selected to represent a wide range of application domains. This variety helps in making the results more generalisable. For example, Commons CLI is a library for processing command-line options, Commons IO is for performing various input/output operations, and Joda-Time is for handling dates and times.
- **Size Diversity:** The applications vary significantly in size, from small applications like Sudoku with 497 lines of code to large ones like Commons Math with over 100,000 lines of code. This size diversity is important because refactorings are applied to both large, well-established projects and smaller, newer projects.
- **Availability of Extensive Test Suites:** The chosen applications come with extensive test suites. These test suites are crucial for executing the applications during the energy consumption experiments and ensuring that the refactorings applied are executed. The presence of comprehensive test suites also ensures that the results are more reliable and that a significant portion of the application's functionality is covered during testing.
- **Test Coverage:** The percentage of statements covered by the test suites for each application was considered, ensuring that the energy consumption tests are sufficiently thorough to exercise the application code where refactorings are applied. The coverage ranges from 63% to 96%, indicating a high level of test coverage across the selected applications.

Sahin *et al.* [8] measured the energy consumption and execution time of these systems before and after each refactoring. They controlled for extraneous variables using automated refactoring tools in Eclipse IDE and employed unit testing frameworks to ensure consistent application behaviour during experiments. The LEAP (Low-power Energy Aware Processing) [59] node was used for detailed power measurements.

The key findings from Sahin *et al.* [8] were:

- **Impact:** Refactorings can significantly affect energy usage, with impacts ranging from a decrease of approximately 4.6% to an increase of approximately 7.5%. Each refactoring therefore has the potential to impact energy usage, though not consistently.
- **Consistency:** The effects of refactorings are not consistent across different applications and platforms.
- **Predictability:** Commonly used metrics like execution time and dynamic execution counts do not accurately predict the energy impacts of refactorings. There was a moderately strong positive correlation between execution time and energy usage (Kendall's tau correlation of 0.81), but execution time alone is insufficient to predict energy usage accurately.

Table 4.3 shows how many times the 9 refactorings were implemented in each of the 6 systems being considered. Each refactoring could only be applied in specific code contexts where the preconditions for this refactoring were met. The clear outlier in this data is the "Extract Local Variable" refactoring, which could not be implemented 4 times in all systems, simply due to lack of suitable locations in the source code.

System	Convert Local Variable to Field	Extract Local Variable	Extract Method	Inline Method	Introduce Indirection	Introduce Parameter Object	**Total**
Commons Beanutils	4	4	4	4	4	4	**24**
Commons CLI	4	0	4	4	4	4	**20**
Commons Collections	4	1	4	4	4	4	**21**
Commons IO	4	2	4	4	4	4	**22**
Commons Lang	4	3	4	4	4	4	**23**
Commons Math	4	4	4	4	4	4	**24**
Joda-Convert	4	0	4	4	4	4	**20**
Joda-Time	4	3	4	4	4	4	**23**
Sudoku	4	0	4	4	4	4	**20**
Total	**36**	**17**	**36**	**36**	**36**	**36**	**197**

Table 4.3: Number of times a Refactoring was implemented per System [8]

Figure 4.3 presents the energy consumption results from Sahin *et al.* [8]. The platform specification key at the bottom of the image, specifying JVM 6 and JVM 7, is not important to discern for the use of this validation, however just shows Sahin *et al.* measured the energy consumption results over two different platforms. Unfortunately, the raw data from these results were over 350 gigabytes in size, so it was not feasible for them to make it publicly available. These values are not required for this validation experiment, however, it is still interesting to view visually, as it shows the range of impact these refactorings have on energy consumption.

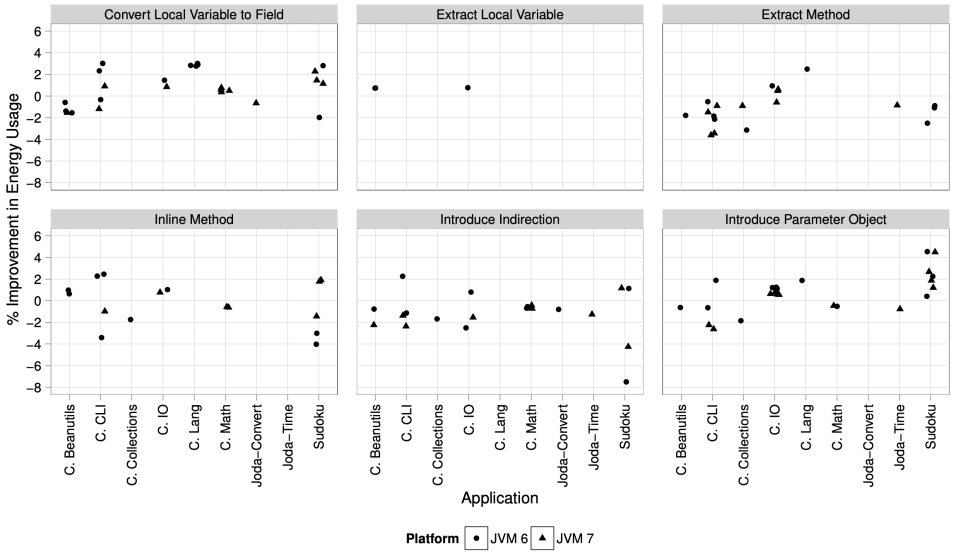


Figure 4.3: Impacts on energy usage of applying refactorings ($\alpha \leq 0.05$) [8].

4.3 Measuring Greenability Metrics and Churn

From these 9 systems, there are two measurements required: Greenability metrics of the original source code, and churn values representing Lines of Code changed between the before and after versions for each refactoring. Both of these measurements are conducted using the tool Sigrid [17]. This tool is explained in Chapter 3 during the selection process of properties and metrics for Greenability. Sigrid is selected for the validation process as it conducts statically analysed source code analysis of Software Quality Models defined in ISO 25010 [10], and follows the methods defined in ISO 5055 [14].

As concluded in Chapter 3, Greenability consists of 7 properties. Given the capabilities of Sigrid, not all metrics can be measured. Figure 4.4 shows which metrics are validated, as well as specifying that the Reliability metrics are represented by a single value rather than individual scores[2]. Therefore 4 properties (rather than 7 properties) and 22 metrics (rather than 48 metrics) are validated.

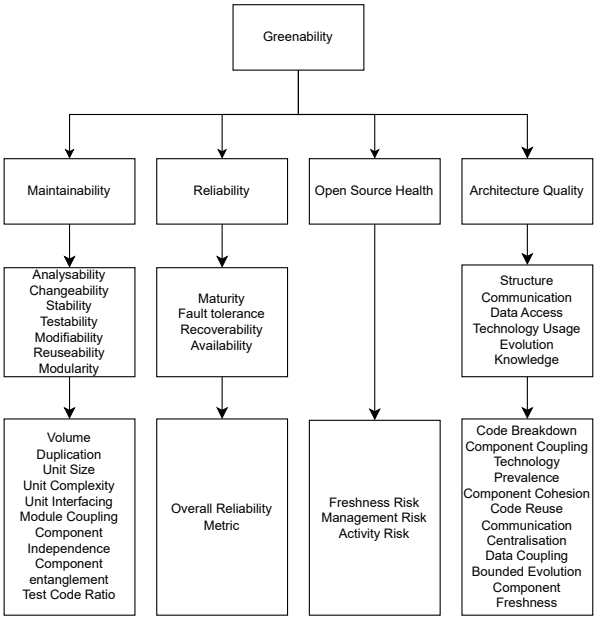


Figure 4.4: Greenability Metrics validated.

[2]This is due to Sigrid's capabilities. It is stated in Future Work that this feature should be updated to provide individual Reliability metric scores.

As described in Chapter 2.4, Sigrid presents all metric results on a scale from 1 to 5 stars. The collection of the Greenability metric scores is implemented through the use of Sigrid's API. For an in-depth explanation, please refer to Sigrid's API documentation[3]. Multiple shell scripts and python scripts are used to upload the systems to Sigrid, and retrieve the metric measurements. These scripts can be found on the online GitHub repository for this project [4].

The systems are uploaded using the following API call:

```
./sigridci/sigridci/sigridci.py --customer ${CUSTOMER} --system ${SYSTEM} --source ${LOCATION} --publish
```

Retrieving the results is performed using the following curl command:

```
URL="https://sigrid-says.com/rest/analysis-results/api/v1/${QUALITY}/${CUSTOMER}/${SYSTEM}"
curl -H "Authorization: Bearer ${SIGRID_CI_TOKEN}" "${URL}" > "$(pwd)/${SYSTEM_BASE}/${SYSTEM_BASE}_${
    QUALITY}.json"
```

Churn values are retrieved using the following POST request:

```
curl -X POST "https://sigrid-says.com/rest/analysis-results/changequality/${CUSTOMER}/${SYSTEM}" \
-H 'content-type: application/json' \
-H "cookie: ssoToken=${SSO_TOKEN}; XSRF-TOKEN=x" \
-H 'x-xsrf-token: x' \
--data-raw '{"startDate":"${START_DATE}","endDate":"${END_DATE}","changeQualityType":"
    NEW_AND_CHANGED_CODE_QUALITY"}' \
| jq . > "churn-${SYSTEM}.json"
```

In practice, the retrieval of these metrics proved slightly more complicated, considering only Maintainability and Architecture Quality metrics can be retrieved from this external API. Reliability metrics are retrieved using an internal API not available to the public. Open Source Health metrics is retrieved through a GET request. These methods require specific tokens, and for privacy reasons, cannot be explained in detail in this report. Results from these measurements are displayed in Chapter 5. These measurements are analysed using Spearman correlation tests.

4.4 Correlation Tests Methodology

This section explains the scripts and tests conducted using the Greenability results and Churn results presented above. Spearman Correlation tests are implemented in R between Greenability metrics and the effort taken to implement each refactoring (represented by churn volume in person months). The scripts are available on the Github repository for this project. The goal of these tests is to prove that higher metric scores result in less effort required to implement refactorings. For example, if a system has less *duplication*, it will be *easier* to refactor because there are fewer instances present in the code base that need to be refactored. This translates into the ideal results being a negative correlation between Greenability scores and Effort per Refactoring.

Two files are needed to conduct the correlation analysis `churn.csv` and `combined_systems.csv`. As a result of running `get_churn.sh`, which is the script created to retrieve the churn results from Sigrid, a CSV file titled `churn.csv` is created with the following layout:

System	Refactoring	totalNewFiles	totalNewVolumeInMonths	totalNewVolumeInLoc
cbeanutils	extractmethod	1	0.537	402.75
cbeanutils	extractvariable	1	0.458	343.5
cbeanutils	inline	1.25	0.789333333	592
...				
sudoku	introducepo	1	0.009333333	7
sudoku	variabletofield	0.5	0.119333333	89.5

Table 4.4: `churn.csv` **results snippet**

As a result of running `get_greenability.sh`, which is the script created to retrieve the Greenability metrics from Sigrid, a CSV file titled `combined_systems.csv` is created with the following layout:

[3] `https://docs.sigrid-says.com/integrations/sigrid-api-documentation.html`
[4] `https://github.com/KirstyGericke/Greenability-Thesis`

Metric	cbeanutils	ccli	ccollections	...	jodaconvert	jodatime	sudoku
Activity Risk	0.5	5.5	5.5		5.5	1.9	5.5
Freshness Risk	0.83	5.5	5.5		5.5	0.5	5.5
Management Risk	1.84	5.5	5.5		5.5	2.09	5.5
...							
unitSize	2.84	2.89	4.21		3.25	3.77	5.3
volume	5.33	5.5	4.95		5.5	4.77	5.5

Table 4.5: `combined_systems.csv` **results snippet**

Using these results, Spearman correlation analysis tests are run in `correlation.R`. This file is fairly long and involves reshaping the data, calculating the Spearman Correlation between every metric and every refactoring's churn value[5] and plotting heat maps to represent these results. The file `correlation_results.csv` contains these results and is included in this project's GitHub repository. The line of code run for the Spearman Test is shown below:

```
...
# Calculate Spearman correlation for each metric against each refactoring's churn values
cor_results <- combined_data %>%
  group_by(Metric, Refactoring) %>%
  summarise(cor_value = cor(MetricValue, Churn, method = "spearman"),
            p_value = cor.test(MetricValue, Churn, method = "spearman")$p.value,
            .groups = 'drop')
...
```

The correlation tests implemented are Spearman's rank correlation coefficient, which measures the strength and direction of association between two ranked variables. It is non-parametric, meaning it doesn't assume a specific distribution for the data. This test is used to assess the relationship between the various software metrics and refactoring churn values. Due to the nature of the data, ties are present, which affects the calculation of exact p-values. As a result, the `cor.test` function in R provides approximate p-values when ties are encountered.

For small sample sizes and data without ties, exact p-values can be calculated. When there are ties, the calculation of exact p-values is more complex and can be computationally intensive or impossible to perform. For larger datasets or when ties are present, approximate p-values are commonly used. These approximations are generally reliable and widely accepted in statistical analysis. In most research contexts, the use of approximate p-values is acceptable, especially when dealing with real-world data where ties are common.

The results from these tests are displayed in Chapter 5. Although tests provide valuable insight into the correlation between Greenability metrics and effort measured by churn values (LOC changed) for various code refactorings, there are threats to validity that need to be addressed. These threats, such as those related to sample size and types of refactorings, can be reduced through an additional source of information: humans. By surveying a group of software developers, effort can be interpreted from a different perspective than LOC changed. Combining results using two perspectives on effort ensures the validity of the Greenability model as well as the accuracy of the correlation results.

4.5 Survey Design

A second validation experiment is conducted on Greenability metrics through a survey distributed to software engineers, developers, and consultants via Google Forms. The goal of the survey is to collect data on the perceived effort required to implement code refactorings, based on their professional experiences. This approach aims to provide subjective insights on the effort associated with Greenability from those directly involved in software development and maintenance.

The goal of this validation is to align the quantitative values calculated for effort, defined as the churn values of lines of code changed during code refactoring, with the effort as perceived by working professionals. Combining these two perspectives on effort allows for a more comprehensive understanding of Greenability, ensuring that the metrics reflect both the actual and perceived effort required for sustainable software refactorings.

The full survey can be found in Appendix B. The survey is structured in the following way:

[5] Volume in PM was used in these correlation tests, however, Volume in LOC would have resulted in the same results.

- **Introduction and Consent:** Participants were informed about the purpose of the survey, defining Greenability as a measure of how easily software can be refactored to improve sustainability. They were assured of the confidentiality and anonymity of their responses.
- **Basic Information:** Questions on participants' familiarity with SIG's quality models, their experience in IT, and their daily responsibilities related to writing, reviewing, and analyzing source code.
- **Refactoring Tasks:** Participants were presented with six common refactoring patterns. For each refactoring, participants were asked to rate the effort required in different systems characterized by five quality attributes: Maintainability, Reliability, Architecture Quality, Freshness, and Performance Efficiency.
- **Effort Ratings:** Participants rated the effort required on a scale from 1 (Less Effort) to 5 (More Effort) for each quality attribute in relation to the given refactoring task.

4.5.1 Refactorings Examples

For each of the six refactorings, simple examples were provided in the form of code snippets. These examples needed to be as short and simple as possible, to avoid confusing or adding unnecessary complexity to the questions. The participants of the survey should be able to understand what code refactoring entails without getting lost in a complex project structure. It was also important that code understanding should not require specific application domain knowledge that the participants have not mastered. The refactoring examples presented to the participants can be found in the Appendix A. Note that this Appendix includes two types of examples per refactoring: Simple and Real Life. Only the Simple examples were used in this survey. The real-life examples were included in this report to illustrate how these refactorings could be applied in more complex, real-world scenarios, providing additional context and depth for further analysis.

4.5.2 Properties Surveyed

This survey presents participants with 5 Greenability properties: Maintainability, Reliability, Architecture Quality, Freshness and Performance Efficiency. The four properties validated by the correlation analysis step were Maintainability, Reliability, Open Source Health and Architecture Quality. It can be seen there are two deviations among these lists: Freshness and Performance Efficiency.

- The term *Freshness* was used in place of *Open Source Health* (OSH) in the survey.
  OSH was measured by Freshness, Management and Activity Risk. As seen in the results in Figure 5.1, the measured churn values for OSH metrics varied much less than other metrics: most systems scored a maximum score of 5.5. This result presents a threat to external validity, since version history was not provided for these systems (resulting in equal Management Risk results) and they were all written in the same language: Java (resulting in equal Activity Risk results). As described in Section 4.1, this data set was not ideal. This implies the comparison of measured and perceived effort for OSH would not be ideal, and rather a sub-category of this property could be presented in the survey: Freshness.
  Freshness was described to the survey participants as a combination of Freshness Risk, Technology Prevalence and Component Freshness. This choice was made considering the commonalities between these three metrics, with the aim of gaining more specific and concise insight into the importance of freshness in refactoring effort from a developer's point of view. These three Freshness metrics were measured by Sigrid and therefore both measured and perceived effort can be compared.
- *Performance Efficiency* was added to the survey to gain extra information about its relevancy. Considering the common correlation between software time behaviour and energy consumption, although it could not be compared to measured effort as the other properties could, it could still provide valuable insights into how developers perceive this relationship.

4.5.3 System Examples

For each of the 5 properties, a table was displayed with three columns: Metrics, System A and System B. Two examples were given for each metric: a value negatively impacting the property score, and a value positively impacting the property score. For example, the *Volume* metric having the value of 10K LOC would result in a higher *Maintainability* score compared to 50K LOC. System A was considered to have a high score for that particular property, hence having good values for all metrics, and System B had a low score. Presenting the properties' metrics in this way removed the need to define each individual metric, allowing the participant to easily recognize what each metric entailed with a practical example.

For example, instead of saying *"Maintainability is measured using seven metrics. Volume is measured in*

terms of lines of code. A smaller volume results in a system having higher maintainability.”, the participant can deduce that information directly from the table. By reducing the amount of text in the survey, the possibility of the participant ignoring or skipping over crucial explanations is reduced, as well as allowing them to focus their cognitive attention on answering the question and reducing the time taken for them to complete the survey as a whole.

Each of the 5 tables provided to the participant to describe each property and their respective metrics are as follows:

Maintainability

Metrics	**System A**	**System B**
Volume	10K LOC	50K LOC
Duplication	No duplicated lines	40% of the codebase is duplicated.
Unit Size	Average method size: 15 lines	Average method size: 50 lines
Unit Complexity	Average McCabe Complexity: 3	Average McCabe Complexity: 10
Module Coupling	Few parameters per method	Many parameters per method
Component Independence	Small interface	Many incoming calls to interface
Component Entanglement	Minimal component communication	High component communication

Reliability

Metrics	**System A**	**System B**
Inter-process Communication	Effective design patterns	Inefficient communication
Logic & Data Flow	Correct boolean logic	Contradicting if statements
Memory Allocation & Release	Properly allocated and freed memory	Memory leaks
Concurrency & Synchronization	Well-designed concurrency	Deadlocks, poor synchronization
Pointers	Proper pointer handling	Frequent null pointer exceptions
Resource Management	Scalable under load	Inefficient, poor scalability
Hard-coded Configuration	Flexible configurations used	Hard-coded configurations
Error Handling	Proper use of try-catch	Frequent unhandled exceptions
Arithmetic Operations	Correct precedence	Integer overflows, division by zero
Dead & Irrelevant Code	Clean and relevant codebase	Many dead code sections

Architecture Quality

Metrics	**System A**	**System B**
Code Breakdown	Classes < 200 lines Methods < 20 lines	Classes > 1000 lines Methods > 100 lines
Component Coupling	Few component dependencies	Many component dependencies
Technology Prevalence	Uses Java 11, Spring Boot 3.0	Uses Java 6, outdated frameworks
Component Cohesion	Follows single responsibility principle	Many responsibilities per component
Code Reuse	No duplication across modules	Duplicate code across modules
Communication Centralization	80% internal centralized code	30% internal code, scattered
Data Coupling	Components use their own databases	Components share a single database
Bounded Evolution	10% co-evolution, changes localized	One change impacts entire code base
Knowledge Distribution	Knowledge spread across 5 people	Concentrated in 1-2 people
Component Freshness	Updated monthly with latest patches	Updated irregularly, every 2-3 years

Freshness

Metrics	System A	System B
Freshness Risk	Dependencies updated within the last 6 months	Dependencies not updated for over 2 years
Component Freshness	Components updated within the last month	Components not updated for over a year
Technology Prevalence	Uses the latest stable versions of technologies	Uses outdated versions of technologies

Performance

Metrics	System A	System B
Time Behavior	Response time < 100ms Handles 1000 requests per second	Response time > 500ms Handles 100 requests per second
Capacity	Can add 1000 more users without affecting performance	Adding 100 more users causes significant slowdowns
Resource Utilization	CPU usage < 50% RAM usage < 4GB under load	CPU usage > 90% RAM usage > 8GB under load

Chapter 5

Results

This chapter presents results for the 9 systems and 6 refactorings, specifically regarding their Greenability Metrics and Properties measured by Sigrid (Section 5.1), Effort Measurement Results (Section 5.2) and Correlation Tests (Section 5.3). All results can be found on the Github repository for this project [1].

5.1 Greenability Results

The results retrieved from Sigrid for the Greenability properties of the 9 software systems gathered from Sahin *et al.* [8] are shown in Table 5.1, and Greenability metrics in Table 5.2. It can be observed that multiple metrics have the value of zero. This absence of data could be due to Sigrid being unable to measure those specific metrics due to lack of version history, considering only one version of the original code was uploaded in this experiment, and in real life systems there realistically would use some version control tool (eg. Github) where code changes over time can be monitored. Another factor, specifically for the *Data Coupling* metric, that no databases were linked and uploaded for these systems, so it is logical for that metric to return no results. The metrics with no findings and hence are not validated in this project are: Component Independence, Component Entanglement, Data coupling, Bounded Evolution and Component Freshness.

This updates the diagram of validated metrics to those seen in Figure 5.1.

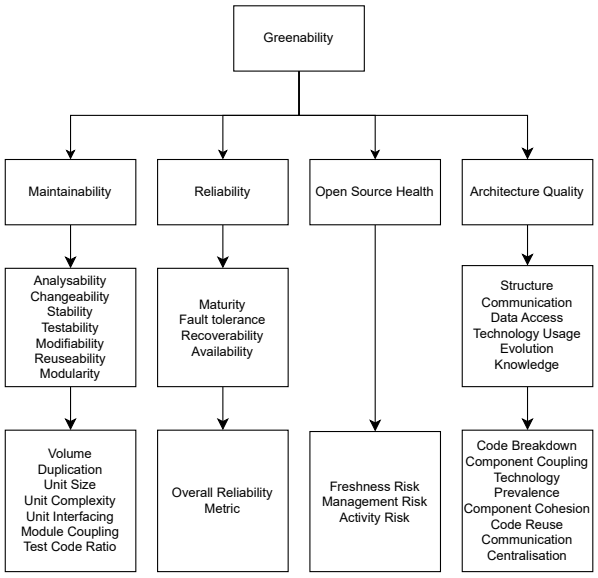


Figure 5.1: Validated Greenability Properties and Metrics

[1] `https://github.com/KirstyGericke/Greenability-Thesis`

System	Maintainability	Reliability	Architecture	Open Source Health
Commons Beanutils	2.6340252	4.4	3.39526	1.060878243
Commons CLI	3.2414644	5.5	4.84503	5.5
Commons Collections	3.7287649	2.2	2.78497	5.5
Commons IO	3.45144368	2.8	4.06133	3.833333333
Commons Lang	3.02291726	3.1	3.9044	2.166666667
Commons Math	2.73814179	3.7	3.58719	5.5
Joda-Convert	4.36361923	5.5	4.94951	5.5
Joda-Time	2.88975561	3.4	3.64217	1.500288087
Sudoku	4.18001535	5.5	5.01658	5.5

Table 5.1: Greenability Property Measurements

Property	Metric	cbeanutils	ccli	ccollections	cio	clang	cmath	jodaconvert	jodatime	sudoku
Maintainability	Volume	5.33	5.50	4.95	5.39	5.06	4.42	5.50	4.77	5.50
	Duplication	2.56	5.39	2.42	3.48	3.13	3.36	4.36	2.92	5.50
	Unit Size	2.84	2.89	4.21	4.06	3.53	2.45	3.25	3.77	5.30
	Unit Complexity	2.34	2.21	3.55	3.07	2.36	2.15	2.49	2.89	5.50
	Unit Interfacing	2.37	2.06	4.38	2.19	2.02	2.58	3.79	3.13	5.12
	Module Coupling	0.88	1.85	3.19	2.21	1.90	1.82	5.50	0.73	0.82
	Component Independence	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00
	Component Entanglement	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00
	Test Code Ratio	2.45	2.48	5.99	2.27	2.03	1.36	1.54	2.52	1.47
Reliability	Reliability	4.40	5.50	2.20	2.80	3.10	3.70	5.50	3.40	5.50
Open Source Health	Activity Risk	0.50	5.50	5.50	5.50	5.50	5.50	5.50	1.90	5.50
	Freshness Risk	0.83	5.50	5.50	0.50	0.50	5.50	5.50	0.50	5.50
	Management Risk	1.85	5.50	5.50	5.50	0.50	5.50	5.50	2.10	5.50
Architecture Quality	Code Breakdown	1.87	1.75	1.29	2.02	1.57	2.35	2.89	1.70	2.73
	Component Coupling	2.23	5.50	1.55	3.59	3.47	2.24	5.50	2.70	5.50
	Technology Prevalence	5.07	5.07	5.16	4.95	4.96	5.06	4.82	4.94	5.18
	Component Cohesion	3.99	5.50	2.62	4.56	4.42	3.50	5.50	3.76	5.50
	Code Reuse	3.68	5.50	3.33	5.50	5.50	5.50	5.50	5.50	5.50
	Communication Centralization	2.50	5.50	1.24	3.18	2.82	2.02	5.50	2.46	5.50
	Data Coupling	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00
	Bounded Evolution	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00
	Component Freshness	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00

Table 5.2: Greenability Metrics Measurements

5.2 Effort Results

Table 5.3 presents the effort, or churn values, for all refactorings implemented in all systems. The unit representing churn in this table is lines of code (LOC). This represents the number of lines of code added and changed during the implementation of a refactoring on a system. This is measured by Sigrid by comparing the source code before and after the refactoring was implemented. Each refactoring was implemented 4 times in each system, therefore the LOC values are decimals, as they were averaged. Table 5.4 presents the churn values converted into Person Months (PM). This represents the number of months it would take the average developer to implement a refactoring on a system. Similarly, the values represent an average of applying the refactoring 4 times in each system. The explanation of Person Months is explained in detail in Section 2.5.

System	Extract Method	Extract Local Variable	Inline Method	Introduce Indirection	Introduce Parameter Object	Convert Local Variable to Field	**Average**
Commons Beanutils	402.75	343.50	592.00	965.25	162.75	80.50	**424.46**
Commons CLI	143.50	0.00	243.75	171.00	87.50	185.00	**138.79**
Commons Collections	308.75	923.00	521.25	2116.00	107.50	533.25	**751.63**
Commons IO	254.25	197.00	1220.75	107.00	203.00	385.25	**394.54**
Commons Lang	853.50	744.00	213.50	2684.75	52.00	153.00	**783.79**
Commons Math	710.50	472.25	2888.75	1083.00	368.75	182.00	**951.04**
Joda-Convert	215.50	0.00	283.50	268.75	6.75	199.50	**162.33**
Joda-Time	422.50	291.33	400.25	1494.50	853.25	463.00	**654.81**
Sudoku	182.50	0.00	111.50	181.00	7.00	89.50	**95.92**
Average	**388.08**	**274.12**	**608.58**	**1030.92**	**205.17**	**252.67**	**460.93**

Table 5.3: Churn in LOC Measurements

System	Extract Method	Extract Local Variable	Inline Method	Introduce Indirection	Introduce Parameter Object	Convert Local Variable to Field	**Average**
Commons Beanutils	0.54	0.458	0.79	1.29	0.22	0.11	**0.57**
Commons CLI	0.19	0.00	0.33	0.23	0.12	0.25	**0.19**
Commons Collections	0.41	1.23	0.69	2.82	0.14	0.71	**1.00**
Commons IO	0.34	0.263	1.63	0.14	0.27	0.51	**0.53**
Commons Lang	1.14	0.992	0.28	3.58	0.07	0.20	**1.34**
Commons Math	0.95	0.63	3.85	1.44	0.49	0.24	**1.27**
Joda-Convert	0.29	0.00	0.38	0.36	0.01	0.27	**0.22**
Joda-Time	0.56	0.39	0.53	1.99	1.14	0.62	**0.87**
Sudoku	0.24	0.00	0.15	0.24	0.01	0.12	**0.13**
Average	**0.51**	**0.33**	**0.85**	**1.34**	**0.27**	**0.34**	**0.61**

Table 5.4: Churn in Person Months Measurements

System	Extract Method	Extract Local Variable	Inline Method	Introduce Indirection	Introduce Parameter Object	Convert Local Variable to Field	**Average**
Commons Beanutils	1.28%	1.09%	1.88%	3.06%	0.52%	0.26%	**1.35%**
Commons CLI	3.03%	0.00%	5.14%	3.61%	1.85%	3.90%	**2.92%**
Commons Collections	0.48%	1.45%	0.82%	3.31%	0.17%	0.84%	**1.18%**
Commons IO	0.99%	0.77%	4.76%	0.42%	0.79%	1.50%	**1.54%**
Commons Lang	1.53%	1.34%	0.38%	4.83%	0.09%	0.28%	**1.41%**
Commons Math	0.52%	0.35%	2.13%	0.80%	0.27%	0.13%	**0.70%**
Joda-Convert	16.36%	0.00%	21.53%	20.41%	0.51%	15.15%	**12.33%**
Joda-Time	0.63%	0.43%	0.59%	2.21%	1.26%	0.69%	**0.97%**
Sudoku	36.72%	0.00%	22.43%	36.42%	1.41%	18.01%	**19.16%**
Average	**6.84%**	**0.60%**	**6.63%**	**8.34%**	**0.76%**	**4.53%**	**4.62%**

Table 5.5: Percentage of Code Refactored

Considering the impact total volume of source code could influence these metrics, Table 5.5 represents the churn values as a percentage of the total source code for a system. This perspective may more accurately represent effort, as normalising effort could reduce the possibility of total volume acting as a confounding variable in these experiments.

5.3 Correlation Results

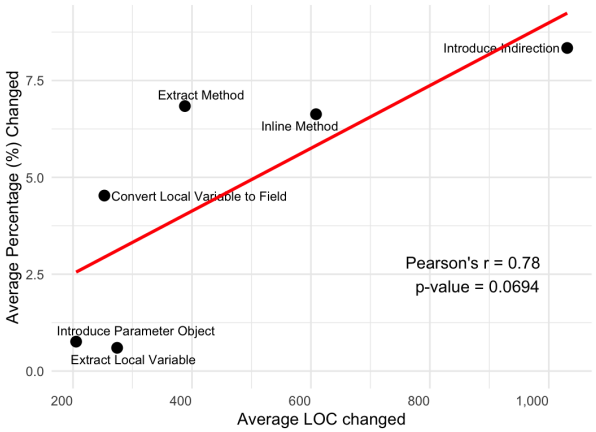
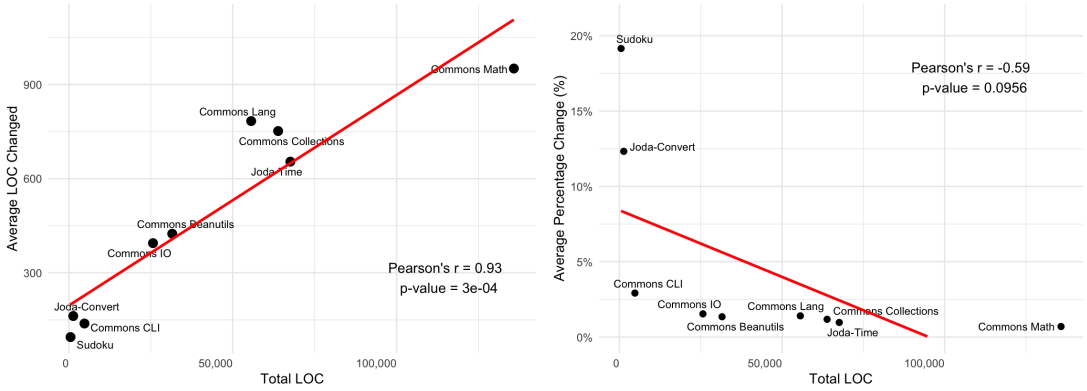


Figure 5.2: Average LOC changed versus Average Percentage Changed per Refactoring



(a) Average LOC changed versus Total LOC

(b) Average % changed versus Total LOC

Figure 5.3: Volume Correlation Tests per System

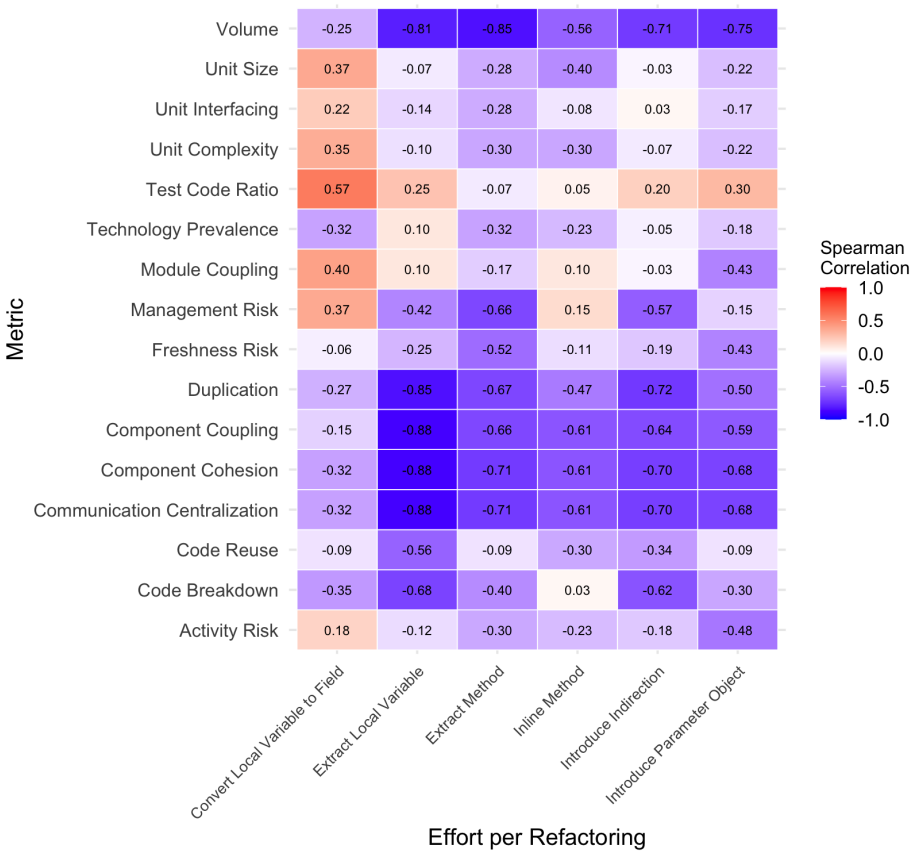


Figure 5.4: Greenability Metrics versus Refactoring Effort Spearman Correlation Heatmap Results

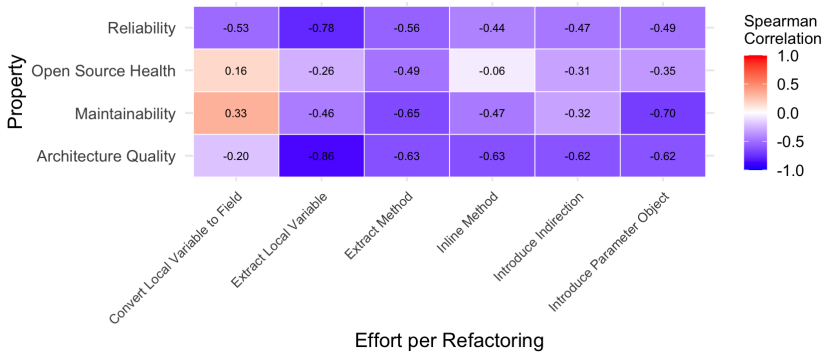


Figure 5.5: Greenability Properties versus Refactoring Effort Spearman Correlation Heatmap Results

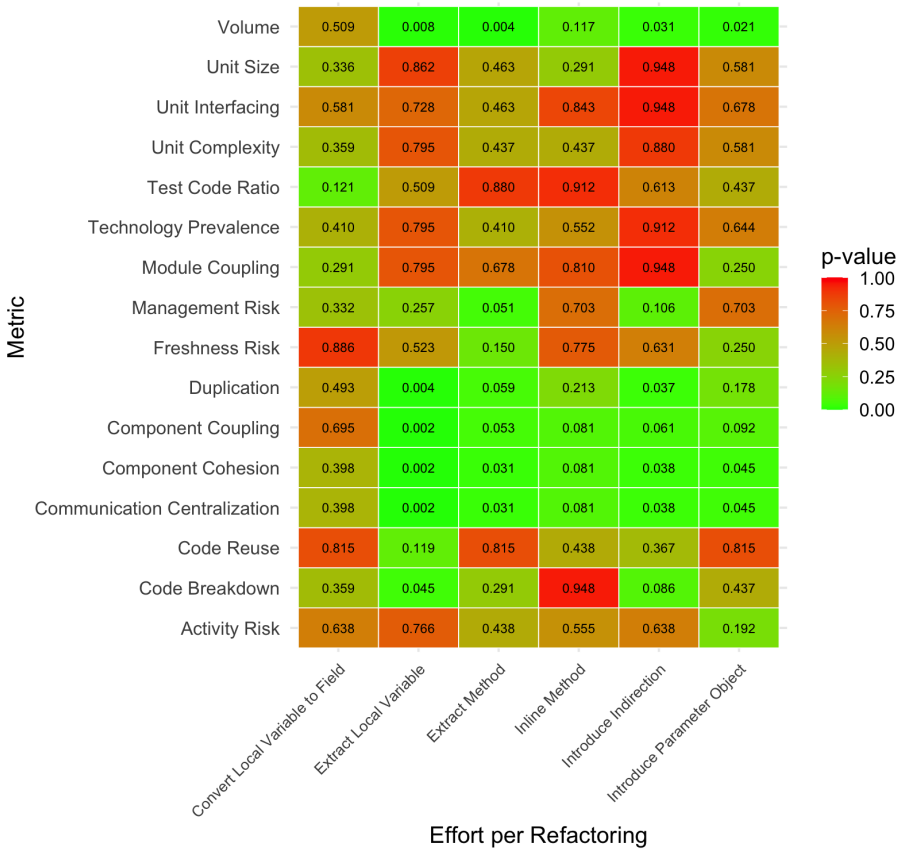


Figure 5.6: Greenability Metrics p-value Heatmap for Correlation results of Figure 5.4.

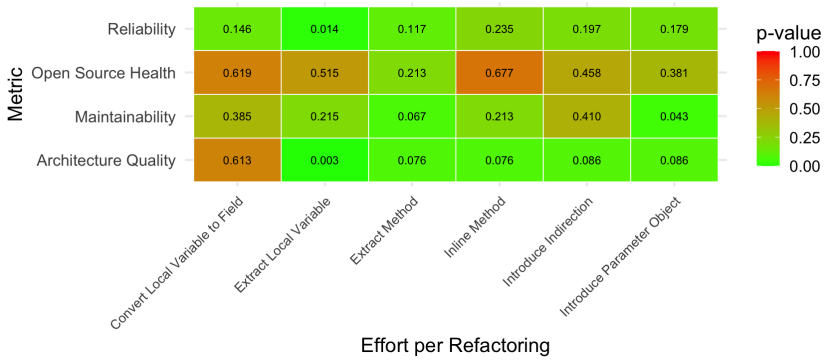


Figure 5.7: Greenability Properties p-value Heatmap for Correlation results of Figure 5.5.

5.4 Survey Results

The results from the survey were collected from Google Forms, automatically converted into `csv` files, and automatically generated graphs for each question. The results can be found on the Github repository [2] for this project. These results contribute to the answer of RQ3: "How does the predictive accuracy of the model compare to developer's perceived refactoring effort?".

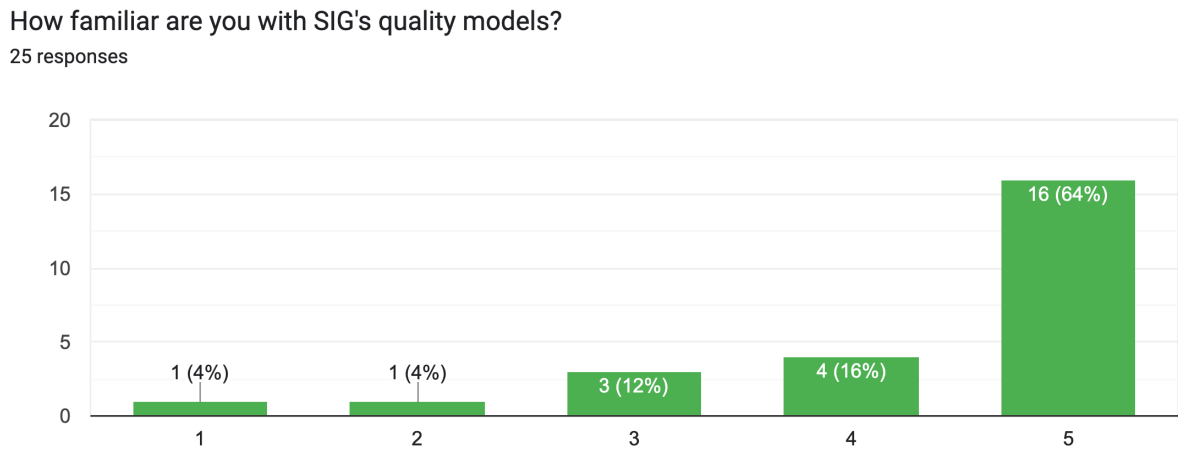


Figure 5.8: Familiarity of participants with SIG's quality models

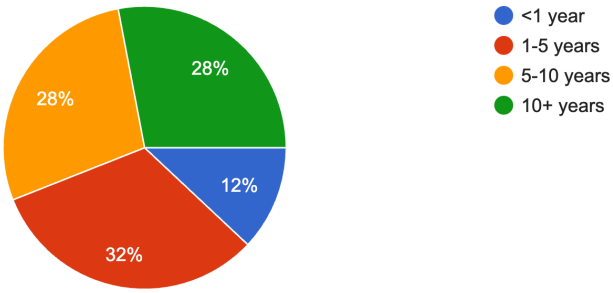


Figure 5.9: Response to "How many years have you been working in IT?"

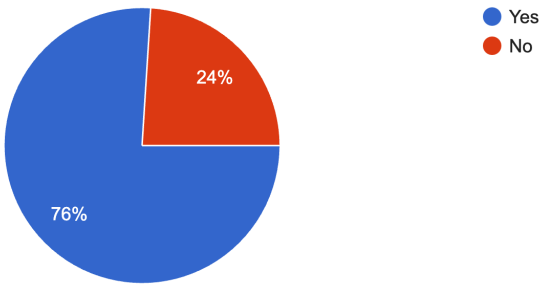


Figure 5.10: Response to "Do your daily responsibilities include writing, reviewing, and/or analyzing source code?"

[2] `https://github.com/KirstyGericke/Greenability-Thesis`

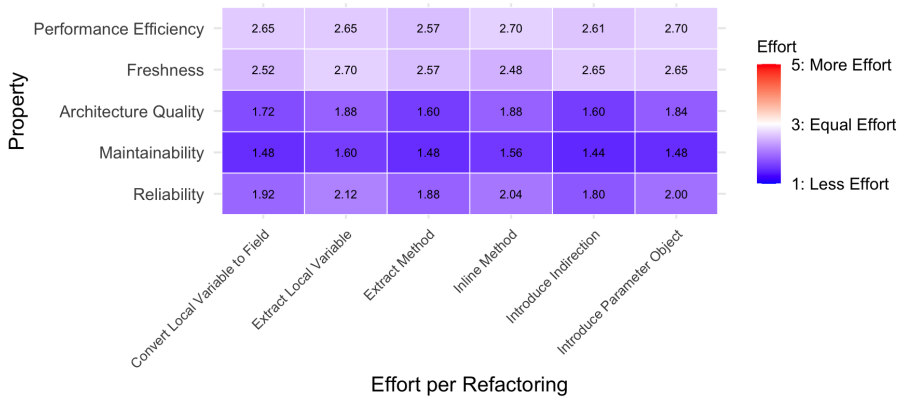


Figure 5.11: Average Perceived Effort Required for Code Refactorings

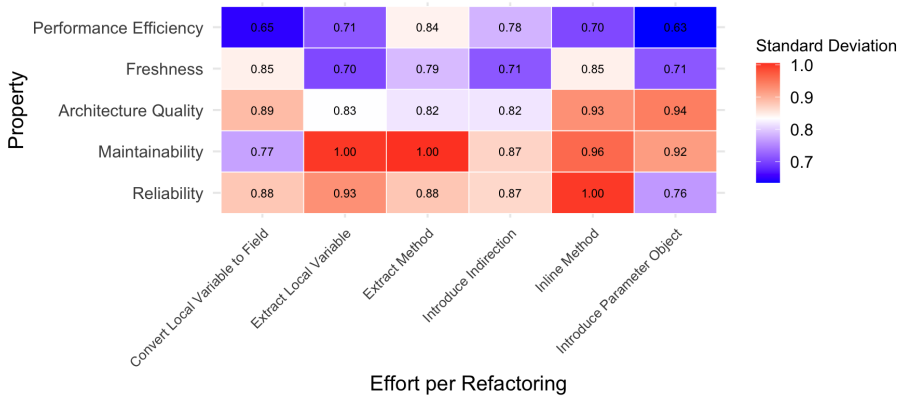


Figure 5.12: Standard Deviation of Perceived Effort Required for Code Refactorings

Standard Deviation provides insights into the level of agreement between survey participants.

5.5 Comparing Correlation and Survey Results

This section presents box plots of all the data already displayed above. This aims to present the results in a graph form rather than heat maps. The results from the survey have been normalised, hence have been transformed from being in a range of 1 to 5, to a range of -1 to 1. This conversion allows a comparison between survey results (perceived effort) and churn correlation results (measured effort).

For the correlation results, -1 represents a negative correlation. In other words, when a property has a higher score, churn will have a lower value, hence improving property scores reduces effort needed to implement refactorings. Similarly, 1 represents a positive correlation, hence improving property scores increases effort needed to implement refactorings.

For survey results, -1 represents less effort is perceived to be needed for systems with higher property scores. 1 represents more effort is perceived to be needed for systems with higher property scores.

It is clear that the concepts represented by -1 and 1 in both these validation experiments are equivalent. For this reason, it is possible to represent the results on a single set of axes. These are seen in Figure 5.13 and Figure 5.14. The individual box plots can be found in Appendix C.

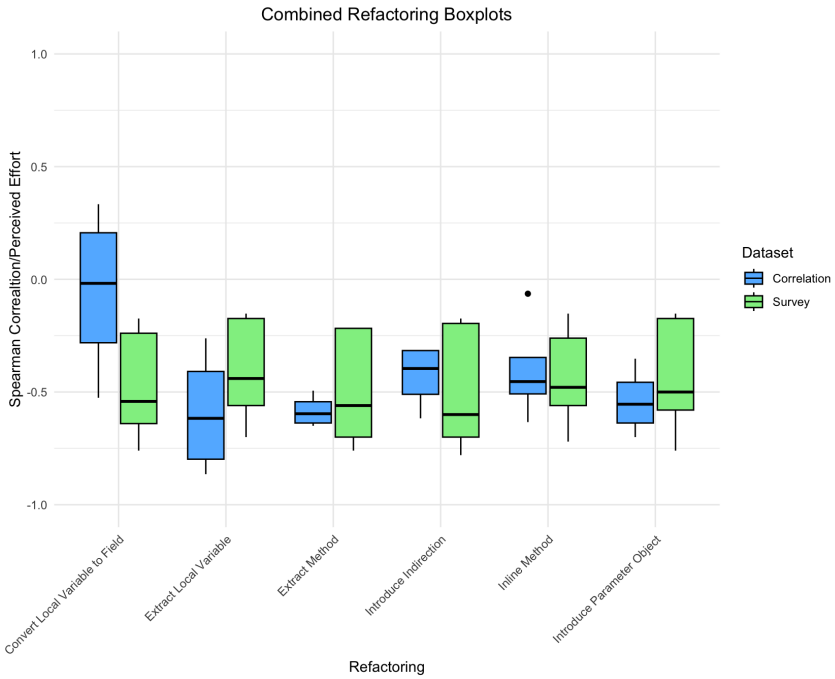


Figure 5.13: Comparing Measured Effort (Correlation tests) to Perceived Effort (Survey) per Greenability Refactoring.

-1 means that a higher score for a property results in less effort needed for a refactoring. 1 means that a higher score for a property results in more effort needed for a refactoring.

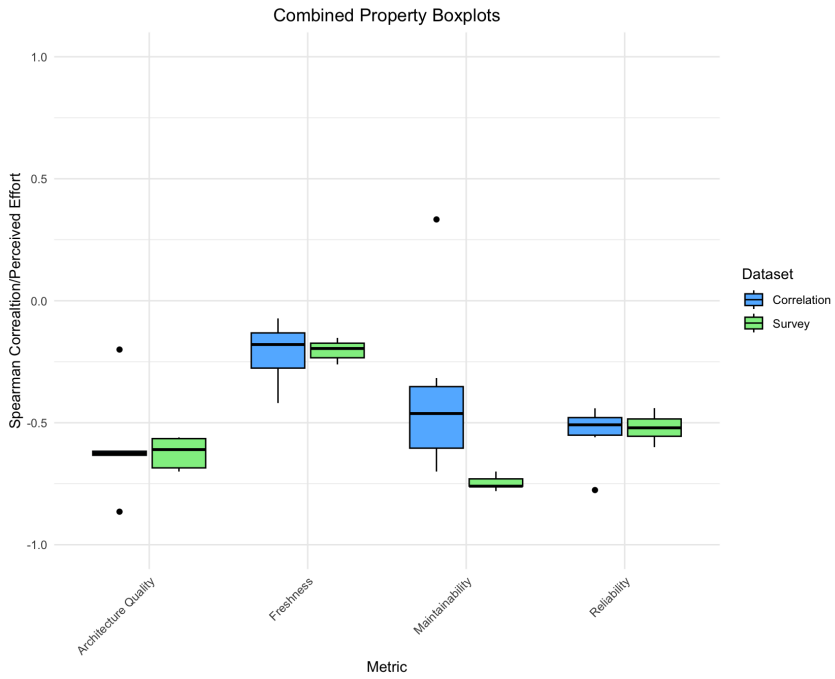


Figure 5.14: Comparing Measured Effort (Correlation tests) to Perceived Effort (Survey) per Greenability Property.

Freshness from Correlation tests accounts *Freshness Risk* and *Technology Prevalence*. *Component Freshness* is not included as Sigrid could not measure this metric, as explained in Section 5.1.

Chapter 6

Discussion

This chapter analyses the results presented in Chapter 5. Section 6.1 discusses the churn results measured for effort, particularly how these relate to the overall volume of a system. Section 6.2 discusses the correlation tests conducted, concluding findings first regarding metrics, then properties, and finally refactorings. Section 6.3 discusses the survey results, particularly regarding whether developers tend to agree or disagree with each other on the perception of effort. Section 6.4 discusses how the measured churn results compare to perceived effort results.

The Greenability metrics measurements shown in Table 5.2 on their own do not provide complete information to analyse. The correlation of these results with churn analysis gives full insight into the Greenability of software projects.

6.1 Effort Analysis

Effort results are presented in Section 5.2, showing 3 tables each representing churn using a different unit: LOC changed, PM changed and percentage of total system volume changed.

Viewing effort as a percentage of the total system's volume that was changed allows for a more accurate analysis of results, as the possibility of a system's volume impacting results is removed. To explore this further, two graphs were created: Average LOC changed vs Total LOC of a system (Figure 5.3a), and Average % changed vs Total LOC of a system (Figure 5.3b). Both graphs show very clear and statistically significant findings: larger systems require more effort to refactor compared to smaller systems, however, this effort is a much smaller percentage of the total system. For example, the biggest system `Commons Math` required the most effort to refactor (951 LOC changed) however this only represents 0.7% of the system. In contrast, the smallest system `Sudoku`, required the least effort to refactor (95 LOC changed), however, this accounted for 19.16% of the total system.

Finding 1: Larger software systems require more effort to refactor compared to smaller systems, however, a lower percentage of the entire system is refactored compared to smaller systems.

When analysing the churn results per refactoring in Figure 5.2, `Introduce Indirection` required on average the most amount of effort to implement, on average changing 8.34% of a system's source code, or on average requiring 1.34 PM to implement. This is due to this refactoring requiring a new class to be created, which other refactorings do not require. `Extract Local Variable` and `Introduce Parameter Object` require the least amount of effort to implement, impacting 0.6% and 0.76%, or taking on average 0.33 and 0.27 PM to implement, respectively. This shows a positive correlation between LOC changed and Percentage changed for Refactorings.

Finding 2: Refactorings that require more effort to implement, also refactor a higher percentage of the overall system's source code.

6.2 Correlation Analysis

Correlation Tests were executed on Greenability Metrics, as well as Greenability Properties (combining the relevant metrics for their respective properties that they define), and the results are presented in Section 5.3. In addition to showing the Spearman correlation values for each metric and property, their p-values are displayed in a heat map to highlight which values are statistically significant (below 0.05).

6.2.1 Metrics Analysis

Figure 5.4 displays all Greenability Metrics, and how an increase in their score affects the effort of 6 code refactorings. These results prove that **on average**, Greenability metrics have a negative correlation with effort. The specification of **on average** is necessary, because out of the 96 combinations of metrics and refactorings[1], 11 have a correlation value above 0.1, indicating a very slight positive correlation. Only 1 combination has a correlation above 0.5: *Test Code Ratio*. In total, 85 combinations of metrics and refactorings have a negative or neutral correlation. This shows that as scores increase for metrics, effort decreases. This validates the Greenability metric selections. For example, a system that has a high (good) score for *component coupling* will require less effort to refactor compared to a system with a lower score.

> **Finding 3:** Higher Greenability Metric scores, on average, reduce the effort required for all code refactorings.

Volume has the highest most consistent negative correlation across refactorings. This correlation is easily explained as a system with a better volume score, hence a fewer more maintainable number of lines of code, has fewer instances to refactor and therefore requires less effort to refactor. Following *Volume*, *Duplication*, *Component Cohesion* and *Communication Centralization* have a strong negative correlation with effort, which is also statistically significant (Figure 5.6). These results justify not only the selection of these metrics but subsequently the Greenability Properties that they belong to: *Maintainability* and *Architecture Quality*.

> **Finding 4:** Volume has the greatest measured impact on ease of code refactorings.

Test Code Ratio has on average the most positive correlation with effort, showing a higher test code ratio results in more effort required to implement refactorings. This result contradicts the selection of this metric for the Software Greenability Model made in Section 3. It is important to consider that this metric represents the ratio between the number of LOC in production code vs testing code, rather than dynamically running each system and measuring the *test coverage* for each system. This representation of testing could be heavily influenced by the overall volume of a system, which could explain these results.

6.2.2 Property Analysis

All Greenability properties on average have a negative correlation with refactoring effort. This shows that improving Greenability Property scores reduces the effort required to implement code refactorings.

The highest positive correlation is 0.33 between *Maintainability* and `Extract Local Variable to Field`. This result is most likely heavily impacted by the high correlation value of 0.57 for `Test Code Ratio`, explained above. This instance is acknowledged as an outlier as all other refactorings have a negative correlation with *Maintainability*: higher scores result in less effort.

Architecture Quality correlation results have two prominent outliers: `Convert Local Variable to Field` and `Extract Local Variable`, as seen in Figure 5.14. Disregarding these, higher Architecture Quality very consistently indicates an improvement in the refactoring effort, as does Reliability. The results for these two properties are also the most statistically significant out of all Greenability Properties (Figure 5.7).

> **Finding 5:** Architecture Quality and Reliability are the Greenability Properties with the biggest measured impact on ease of refactorings.

[1] 16 Greenability Metrics * 6 Code Refactorings = 96 combinations

6.2.3 Refactoring Analysis

The refactoring `Convert Local Variable to Field` has the largest standard deviation in Correlation results, as well as the highest average correlation of 0, indicating on average Greenability properties do not influence the ease at which this property is implemented.

All other refactorings have an average negative Spearman correlation with both Greenability metrics and properties, as seen in Figure 5.13. This proves that on average, better Greenability scores reduce effort needed for all refactorings.

The refactoring `Extract Local Variable` has on average the strongest negative correlation, and was the most statistically significant, implying better Greenability metric scores make this refactoring much easier to implement. These results are most likely caused by the fact that this refactoring was implemented only 17 times instead of 36 like the other refactorings. The power of a statistical test is its ability to detect an effect if there is one, and with a reduced sample size of this particular refactoring, the statistical power is lower compared to other refactorings. In smaller samples, each individual data point has a greater influence on the overall result. This can make the p-value more sensitive to the specific data points included in the analysis. A smaller sample size also often results in higher variability in the data. This can lead to less precise estimates of the correlation, which can affect the significance of the results. Despite the significant results for the `Extract Local Variable` refactoring, it's important to consider the impact of the smaller sample size when interpreting these findings. The reduced number of implementations may contribute to the observed statistical significance, and therefore these results can not be generalised.

6.3 Survey Analysis

The basic questions show that 64% of participants are familiar with SIG's quality models, and 76% of participants write, review and/or analyse code as part of their daily tasks. The experience of participants is distributed quite evenly between 1 and 10 years of experience in IT.

The survey aimed to gather developers' opinions on whether a better score for various properties would make refactorings easier, using a scale of 1 to 5. The questions were phrased as "Would implementing this refactoring in System A require more or less effort than in System B?" given System A had a better score for a certain property. The properties evaluated include Maintainability, Reliability, Architecture Quality, Freshness, and Performance Efficiency. The refactorings evaluated were the same as presented in the results from the correlation tests, from Sahin *et al.* [8]: Convert Local Variable to Field, Extract Local Variable, Extract Method, Introduce Indirection, Inline Method and Introduce Parameter Object.

6.3.1 Perceived Effort and Standard Deviation Analysis

For all Greenability Properties and all code refactorings, developers consistently perceive higher property scores result in less effort being required for code refactorings.

The standard deviation results highlight the areas where developers have diverse opinions and where there is more consensus. High variability in certain refactorings suggests that individual experiences and contexts play a significant role in shaping developers' perceptions of effort. For example, the high standard deviation in `Extract Method` and `Extract Local Variable` for maintainability suggests that these refactorings can be straightforward for some developers while being complex for others, likely depending on the specific context of the code base and the developer's experience.

Conversely, lower variability in refactorings such as `Convert Local Variable to Field` and `Introduce Parameter Object` indicates more uniform experiences among developers, suggesting that these refactorings might be more predictable and consistently understood across different contexts.

The standard deviation of 1.0 for *Maintainability* for *Extract Method* and `Extract Local Variable`, and for *Reliability* and `Inline Method` indicates the highest variability in responses. This suggests that developers had mixed opinions on the effort required for these refactorings, with some finding them very easy and others finding them quite challenging. Despite this lack of consensus, on average higher *Maintainability* is perceived to reduce refactoring effort the most.

Finding 6: Maintainability is perceived by developers to have the highest impact on code refactoring effort, however, opinions are dispersed.

Performance Efficiency and *Freshness* had the lowest deviation in responses and highest perceived effort values. This shows these two properties have a very consistent perception of not impacting the ease of code

refactorings.

> **Finding 7:** Performance Efficiency and Freshness are perceived by software developers as having a lesser impact on refactoring effort compared to other Greenability Properties.

6.4 Survey vs Correlation

Figure 5.14 and Figure 5.13 show box plots for both correlation and survey results side by side, in order to analyse the difference between two measurements of effort: churn (LOC changed) and perceived effort. The comparison of these are as follows:

- **Properties:** Developers' perception of effort aligned accurately with the measured churn results for *Reliability, Freshness* and *Architecture Quality*. On average, developers overestimate how much easier refactorings would be with higher *Maintainability* scores.

  > **Finding 8:** Developers' perception of effort aligned accurately with measured effort results for *Reliability, Freshness* and *Architecture Quality*

  > **Finding 9:** Developers overestimate how much easier refactorings would be on systems with higher *Maintainability* scores.

- **Refactorings:** Developers' perception of effort:
  - aligned accurately with `Extract Method, Inline Method` and `Introduce Parameter Object`.
  - overestimated *Convert Local Variable to field* and *Introduce Indirection*
  - underestimated `Extract Local Variable`

The correlation tests and survey all have **average** values for all refactorings and properties below zero, indicating that all refactorings would be easier to implement by improving any Greenability property. The **average** specification in this sentence is important, as there are outliers present in the churn results.

> **Finding 10:** On average, all refactorings are easier to implement if any Greenability Property is improved. This is true for both measured and perceived effort.

These box plots collectively highlight the relationship between Greenability properties and refactoring techniques with both perceived and measured effort. The survey results, normalized to a range of -1 to 1, offer insight into perceived effort, while the correlation results, representing actual measured effort in terms of lines of code changed, provide a quantitative measure. This comparison helps to understand the impact of Greenability metrics on software refactoring efforts, validating Greenability Metrics. The visual representation supports the impact of different properties and refactorings, which can contribute to future improvements in software engineering practices.

6.5 Summary of Findings

There were 10 findings concluded in this discussion. Each of these contributes to answering the four research questions of this project. The relation between findings and research questions is presented in Table 6.1. Each question is stated and answered using these findings in the Conclusion in Chapter 8.

Number	Finding	Research Question
Finding 1	Larger software systems require more effort to refactor compared to smaller systems, however, a lower percentage of the entire system is refactored compared to smaller systems.	RQ4
Finding 2	Refactorings that require more effort to implement, also refactor a higher percentage of the overall system's source code.	RQ4
Finding 3	Higher Greenability Metric scores, on average, reduces the effort required for all code refactorings.	RQ2
Finding 4	Volume has the greatest impact on ease of code refactorings.	RQ2
Finding 5	Architecture Quality and Reliability are the Greenability Properties with the biggest measured impact on the ease of refactorings	RQ1
Finding 6	Maintainability is perceived by developers to have the highest impact on code refactoring effort, however, opinions are dispersed.	RQ3
Finding 7	Performance Efficiency and Freshness are perceived by software developers as having a lesser impact on refactoring effort compared to other Greenability Properties.	RQ3
Finding 8	Developers' perception of effort aligned accurately with measured effort results for Reliability, Freshness and Architecture Quality.	RQ3
Finding 9	Developers overestimate how much easier refactorings would be on systems with higher Maintainability scores.	RQ3
Finding 10	On average, all refactorings are easier to implement if any Greenability Property is improved. This is true for both measured and perceived effort.	RQ1

Table 6.1: Findings Answering Research Questions

6.6 Threats to validity

This section assesses the threats to validity of this project, specifically construct, internal and external validity.

6.6.1 Construct Validity

Construct Validity refers to whether the variables in the study accurately reflect the constructs they are intended to measure. This is accounted for by ensuring the definitions for all properties and metrics used in Greenability align well with the theoretical constructs they represent by following standardised ISO definitions [10, 13, 14], as well as using a tool, Sigrid, to measure these values. Sigrid is created by an ISO-certified company, SIG, and follows methods of measurement defined in ISO 5055 [14]. A threat to construct validity in this study could potentially relate to *effort*.

Effort is a complex topic in the field of software engineering as it can be impacted heavily by the skills, experience and working styles of developers. An additional difficulty is introduced when measuring effort statically rather than dynamically. Section 2.5 describes in detail how effort is measured statically in terms of Person Months.

An additional consideration is whether it is logically sound to measure *effort* of refactorings that can be automated, considering there is no manual effort from a human being implemented. This highlights yet another complexity of defining the concept of *effort* in software development. When it comes to measuring effort using static source code analysis represented by churn as lines of code changed, it is clear it doesn't matter whether it was implemented automatically or manually: the same number of lines of code had to be changed. If effort was rather being measured by time, then obviously manual implementation would affect results significantly. Determining the validity of churn representing effort is a complex research topic out of the scope

of this project. It is assumed that since ISO 5055 [14] and SIG use this metric as a measurement of effort, it is valid enough to be used in this project.

Finally, careful thought was taken to not define Greenability as a *predictor*, but rather an *indicator*. This is because the terms "predictor" and "indicator" can carry different connotations. A "predictor" suggests a variable that can forecast or anticipate future outcomes, implying a stronger, possibly causal, relationship. An "indicator", on the other hand, may simply suggest a sign or measurement of the current state without implying causality. Using precise and consistent terminology helps in accurately conveying the constructs being studied, thus enhancing the construct validity of this project. Using the term "predictor" introduces implications for internal validity as well. By suggesting a predictive relationship, it is necessary to acknowledge the necessity to control for confounding variables that could influence this relationship.

6.6.2 Internal Validity

Internal Validity refers to the extent to which the results of a study can be attributed to the treatments used in the study, rather than other factors. It deals with the causal relationships between variables.

- **Confounding Variables:** The presence of other variables that could influence the relationship between software quality attributes and Greenability can threaten internal validity. The biggest potential confounder could be a system's overall volume. This is accounted for by calculating churn values as a percentage of total volume, however, it is acknowledged throughout this paper that volume could influence results.
- **Selection Bias:** The selection procedure for repository selection is described in detail in Section 4.1. The selection process for participants answering the survey experiment involved selecting developers who had sufficient knowledge to provide accurate responses. To confirm the participants' level of expertise, 3 questions were added to the beginning of the survey regarding their familiarity with the models used in this project, their years of experience and their daily tasks.

6.6.3 External Validity

External Validity refers to the extent to which the findings can be generalised to other settings, populations, times, and measures.

- **Sample Representativeness:**
  - The repository used in the validation of this study could propose a threat to external validity due to the small sample size of 9 systems and 6 refactorings being measured. In reality, since each of these refactorings was implemented 4 times, there were 197 files analysed. This amount appears statistically significant, however considering these 4 refactorings were averaged, only 54 data points were presented. Unfortunately, it was not possible to find a larger data set, as explained in the selection process in Section 4.1.
  - The refactorings used did not strictly *decrease* energy consumption, but rather *increase and decrease* as concluded by Sahin *et al.* [8]. This proposes the threat that although it is proven for Greeenability properties to decrease refactoring effort if the refactorings do not reduce energy consumption, the results may not be generalisable to all green software patterns.
  - The 9 systems were diverse in their metric scores, volume sizes and test coverage values, which reduces that threat to external validity.
  - The 6 refactorings were the most commonly used refactorings, implying they are widely used not only in common developer refactoring tasks but specifically green refactorings.
  - The survey was conducted on 25 participants. This could pose a threat to generalisability and would have preferably had a larger sample size.
- **Setting and Context:** The properties selected for Greenability were ensured to be applicable to multiple software frameworks, languages and tools, however using static analysis could reduce the generalisability of Greenability metrics in contexts that require dynamic analysis.

Chapter 7

Related work

This section provides additional literature on topics related to this project, including Testability in Section 7.1 and Performance Measurement in Section 7.2. Energy Consumption Measurement metrics are listed in Section 7.3, and the differences between hardware and software energy measurement tools and models are clarified in Section 7.4. Finally the impact programming languages can have on software sustainability is discussed in Section 7.5.

7.1 Testability

Testability is mentioned in this project as a sub-characteristic of Maintainability, and is measured by Sigrid using *Test Code Ratio*. Testability is an important concept in software development as it ensures that software can be easily and efficiently tested to identify and fix defects, thereby improving the overall quality and reliability of the software. This section covers literature on testability and how it relates to software sustainability.

Testability is informally the probability that a program will fail under test if it contains at least one fault [60]. ISO/IEC 25010 [10] define testability as *"attributes of software that bear on the effort needed to validate the software product"*, as well as being a sub-characteristic of maintainability. Testing involves ensuring several quality attributes are met, including performance, security and usability [61]. It can increase complexity, resource and time consumption [33], as well as can contribute up to 50% of software development cost and effort [62].

Beer *et al.* [33] describes how sustainable test processes can significantly improve software development's environmental, social, economic, and technical sustainability.

Leitner and Bezemer [34] presents an exploratory study on the practices of performance testing in Java-based open-source projects. Common practices found include the limited popularity of writing performance tests, the small size of performance test suites, the lack of standardized organization for performance tests, the diversity of test types, and the limited adoption of performance testing frameworks. Understanding these challenges and practices around performance testing emphasises how testability impacts software sustainability, especially regarding maintainability and evolution over time.

Rosado de Souza *et al.*[35] categorise Software Sustainability by intrinsic (internal software qualities) and extrinsic (cultural and organisational) factors. Intrinsic factors include modularity, encapsulation and testability, while extrinsic factors include how software is resourced, supported and shared. Although Rosado de Souza *et al.* conclude testing is an important consideration for sustainability, the research method used semi-structured interviews on a small sample group of developers with no significant analysis performed.

Verdecchia *et al.* [63] proves a shift from brute force to intelligent testing strategies result in a significant reduction in both test cases and energy consumption. By implementing methodologies like static code analysis, Test-Driven Development (TDD), and AI-based validation, testing became more efficient and effective. This approach also enhanced code quality and reduced the environmental impact of testing. The transformation resulted in improved test coverage, better resource utilization, and a *"double-digit reduction in electrical energy use"*.

7.2 Dynamic Analysis of Performance

Performance Efficiency was included in the Greenability model for its applicability, however could not be validated due to the capabilities of Sigrid. Although there are statically measurable metrics that can be used for software performance, dynamic analysis methods are particularly effective for performance evaluation in software systems because they allow for measurements under real world conditions, which static analysis cannot fully replicate [64]. Dynamic analysis helps uncover hard-to-find bugs, such as memory leaks and concurrency issues, by analysing the software during its execution rather than just inspecting the source code. Dynamic measurement metrics, collected from Cortellessa *et al.* [65], include:

- Time Behaviour
  - **Response Time:** The time taken for the system to respond to a request. This can be measured as Average Response Time or 90th or 95th Percentile Response Time.
  - **Throughput:** The number of transactions or requests processed per unit of time, or in other words, Requests Per Second (RPS) or Transactions Per Second (TPS).
  - **Latency:** The delay between a request and the start of a response.
  - **Service Time:** The time required to complete a single transaction or request.
- Capacity
  - **Scalability:** The ability of the system to handle increased load without performance degradation. This can be measured as Number of Concurrent Users/Connections or Load Test Results.
  - **Peak Load Handling:** The system's performance under maximum expected load. This can be measured as Maximum Transactions Per Second (TPS) under Peak Load, or System Saturation Point.
  - **Load Thresholds:** The maximum capacity the system can handle while meeting time behavior requirements. This can be measured as Load at which Response Time Degrades or Load at which Throughput Peaks.
- Resource Utilization
  - **CPU Utilization:** The percentage of CPU capacity used. This can be measured as Average CPU Utilization or CPU Utilization under Peak Load
  - **Memory Usage:** The amount of memory used by the system. This can be measured as Average Memory Utilization or Peak Memory Utilization
  - **Disk I/O:** The rate of read/write operations on the disk. This can be measured as Disk Read/Write Operations Per Second or Disk Throughput (MB/s)
  - **Network Utilization:** The amount of network bandwidth used. This can be measured as Network Throughput (Mbps) or Network Latency
  - **Resource Contention:** The degree to which different processes or threads compete for system resources. This can be measured as Context Switches Per Second or Wait Time for Resource Access.

7.3 Measuring Energy Consumption

Table 7.1 presents metrics mentioned in three literature sources which measure software energy consumption, as well as their definitions. This table provides a good example of the wide range of measurements that could be taken to calculate software energy consumption.

Existing tools that measure software energy consumption include Intel's Energy Checker SDK [66], which relates energy consumption to useful units of work. Microsoft's Joulemeter [67] can measure the energy consumption of software applications running on Windows platforms, however although it is powerful, it is limited to specific platforms and to non-distributed software systems. Python's CodeCarbon [68] is *"a lightweight software package that estimates the amount of carbon dioxide produced by the cloud or personal computing resources used to execute the code"*. It also gives developers recommendations for code optimisation.

Paper	Category	Abbrev.	Metric	Definition
[4]	Energy Behaviour	ACC	Annual Component Consumption	The annual energy consumption per application component measured in kWh.
		RIC	Relative Idle Consumption	The percentage of annual idle energy consumption to total per application component.
		CCUW	Component Consumption per Unit of Work	The average energy consumption (in kWh) of each software component per unit of work delivered.
	Capacity	PGR	Peak Growth	A percentage of the actual power demand of application components to the maximum power that hardware can provide during peak workload.
		PRO	Provisioning	A percentage of the energy usage of application components to the maximum energy budget of the hardware.
	Resource Utilization	CNS	Consumption Near Sweet-spot	A percentage that shows the efficiency of the application with respect to its optimal efficiency when delivering work units [69].
		PG	Proportionality Gap	Difference between ideal power provisioning and actual provisioning during average application utilization [70].
		OPO	Operational Overhead	A percentage that denotes the energy overhead required by the infrastructure to process application component workloads.
[69]		AE	Annual consumption	The total energy consumption of an e-service on an annual basis.
		ET	Consumption per transaction	The average energy consumption per executed end-user or business transaction.
		RE	Relative efficiency	The efficiency of the e-service with respect to its optimal efficiency (typically at maximal load).
[70]		DR	Dynamic Range	A first order approximation for energy proportionality.
		EP	Energy Proportionality	A function of the dynamic range and the linearity of the energy proportionality curve.
		LD	Linear Deviation	The measure of the energy proportionality curve's linearity.
		PG	Proportionality Gap	The measure of deviation between the server's actual energy proportionality and the ideal energy proportionality at individual utilization levels.

Table 7.1: Energy Consumption Measurement Metrics

7.4 Granularity and Software vs Hardware Measurements

Granularity is a concept used by many relevant literature sources, which describes the level of detail or scale at which something is examined or measured. In the context of software energy measurement, granularity refers to with the structural levels of software (software granularity) and the levels of hardware resources (hardware granularity) [71].

- **Software** levels of granularity, from least to most granular, are: Operating System (OS) Level, Application Level, Process Level, Thread Level, Method/Procedure Level, Line of Code and Instruction Level.
- **Hardware** levels of granularity are: CPU, Memory, HDD/Storage, Network, Screen and Whole Machine

Granularity is important because it affects the accuracy and relevance of energy consumption data. Different levels of granularity provide insights into different aspects of energy usage. Both high and low granularity have advantages depending on the goal of the measurements. More detailed and fine-grained measurements (e.g. instruction level or line of code) are essential for understanding specific sources of energy consumption within software. Broader measurements (e.g., application level or whole machine) are useful for general energy profiling, understanding overall system efficiency, and making high-level decisions about software and hardware design [71].

In practice, energy measurement tools and methods vary in granularity. Ghaleb [71] presents on overview of different energy consumption measurement instruments, and how each method can measure a specific level of granularity either with respect to software or hardware.

- **Software** Tools include simulators, profiling tools, and system monitoring tools that estimate energy consumption based on software execution. They are generally more cost-effective and flexible but may involve some estimation errors.
- **Hardware** Tools include specialized computers and power meters that provide precise measurements of actual power usage. They are more accurate but often more expensive and less accessible.

Figure 7.1 presents a comprehensive summary of various methods and tools used to measure energy consumption and their respective software and hardware granularity levels. Please refer to Ghaleb [71] for more information about these tools.

Methods Type		Measurement method	Sampling frequency	Software granularity levels						Hardware level of details						
				OS	Application	Process	Thread	Method/Procedure	Line of Code	Instruction	CPU	Memory	HDD	Network	Screen	More	Whole machine
Software Tools	Architecture simulators	ARMmulator extension [4]	Every clock cycle		✓						✓	✓				✓	
		ARMmulator extension [14]	Every clock cycle					✓			✓	✓				✓	
		Wattch [15]	Every clock cycle		✓						✓	✓					
		SoftWatt [16]	Every clock cycle	✓	✓						✓	✓	✓				
	Application profiling tools	pTop/pTopW [17]	T Hz (adjustable: T <1 Hz)			✓					✓		✓	✓			
		Jolinar [7]	2 Hz		✓						✓	✓	✓				
		Jalen [18]	100 Hz		✓		✓	✓			✓	✓	✓				
	System monitoring tools	PowerAPI [19]	2 Hz (adjustable)	✓		✓					✓	✓	✓				
		PowerTop [20]	1 Hz	✓		✓					✓			✓	✓	✓	
		Joulemeter [21]	1 Hz	✓	✓						✓	✓	✓		✓		
		Intel Power Gadget [6]	10 Hz (adjustable: 1-1000 Hz)	✓				✓			✓						
	Program analysis tools	PPROF [26]	Every clock cycle		✓					✓	✓						
Hardware Devices	Specialized Computers	LEAP) node [2]	10 Hz		✓			✓			✓	✓	✓				
		Spartan-6 FPGA [30]	5 Hz		✓						✓			✓	✓	✓	
	Power Meters	Watts Up? Pro [3]	1 Hz	✓	✓												✓
		Monsoon [33]	5 KHz	✓	✓												✓
		Arduino Uno [34]	500 kHz		✓												✓
Hybrid Methods	Software-Software	e-Surgeon [36]	100 Hz			✓	✓	✓			✓	✓	✓	✓			
	Software-Hardware	PowerScope [37]	100 kHz			✓		✓									✓

Figure 7.1: Comparison of energy measurement methods and their associated level of granularity [71].

Please ignore the references in this figure, as they are applicable to the paper Ghaleb [71].

7.5 Impact of Programming Languages on Energy Consumption

Pereira *et al.* [72] conducted an in-depth study on the runtime, memory usage, and energy consumption of 27 well-known software languages. They use the Computer Language Benchmarks Game (CLBG) to gather comparable implementations of diverse programming problems in multiple languages. The energy consumption was measured using Intel's Running Average Power Limit (RAPL) tool, while execution time and memory usage were measured using standard tools in Unix-based systems. Each solution was executed and measured ten times to ensure accuracy and consistency.

Results from this study are shown in Figure 7.2. The results reveal how slower/faster languages consume less/more energy and the influence of memory usage on energy consumption. It can be clearly seen that C and Rust significantly outperformed other languages in energy efficiency. In fact, they were roughly 50% more efficient than Java and 98% more efficient than Python. These findings can be used to help software engineers decide on the best language to use when energy efficiency is a priority. Pereira *et al.* conclude that understanding the energy efficiency of programming languages is complex and context-dependent. They provide valuable data and methodologies for further research.

Total					
	Energy		Time		Mb
(c) C	1.00	(c) C	1.00	(c) Pascal	1.00
(c) Rust	1.03	(c) Rust	1.04	(c) Go	1.05
(c) C++	1.34	(c) C++	1.56	(c) C	1.17
(c) Ada	1.70	(c) Ada	1.85	(c) Fortran	1.24
(v) Java	1.98	(v) Java	1.89	(c) C++	1.34
(c) Pascal	2.14	(c) Chapel	2.14	(c) Ada	1.47
(c) Chapel	2.18	(c) Go	2.83	(c) Rust	1.54
(v) Lisp	2.27	(c) Pascal	3.02	(v) Lisp	1.92
(c) Ocaml	2.40	(c) Ocaml	3.09	(c) Haskell	2.45
(c) Fortran	2.52	(v) C#	3.14	(i) PHP	2.57
(c) Swift	2.79	(v) Lisp	3.40	(c) Swift	2.71
(c) Haskell	3.10	(c) Haskell	3.55	(i) Python	2.80
(v) C#	3.14	(c) Swift	4.20	(c) Ocaml	2.82
(c) Go	3.23	(c) Fortran	4.20	(v) C#	2.85
(i) Dart	3.83	(v) F#	6.30	(i) Hack	3.34
(v) F#	4.13	(i) JavaScript	6.52	(v) Racket	3.52
(i) JavaScript	4.45	(i) Dart	6.67	(i) Ruby	3.97
(v) Racket	7.91	(v) Racket	11.27	(c) Chapel	4.00
(i) TypeScript	21.50	(i) Hack	26.99	(v) F#	4.25
(i) Hack	24.02	(i) PHP	27.64	(i) JavaScript	4.59
(i) PHP	29.30	(v) Erlang	36.71	(i) TypeScript	4.69
(v) Erlang	42.23	(i) Jruby	43.44	(v) Java	6.01
(i) Lua	45.98	(i) TypeScript	46.20	(i) Perl	6.62
(i) Jruby	46.54	(i) Ruby	59.34	(i) Lua	6.72
(i) Ruby	69.91	(i) Perl	65.79	(v) Erlang	7.20
(i) Python	75.88	(i) Python	71.90	(i) Dart	8.64
(i) Perl	79.58	(i) Lua	82.91	(i) Jruby	19.84

Figure 7.2: Normalized results for Energy, Time, and Memory consumption of programming languages [72]

Chapter 8

Conclusion

This thesis investigates the relationship between software quality attributes and software "Greenability", which is the ability of software to become more sustainable through the ease of implementing green software refactorings that reduce software energy consumption. The Greenability Model is summarised in Figure 8.1. The first row lists Greenability Properties, the second sub-characteristics, and the bottom row lists Greenability metrics.

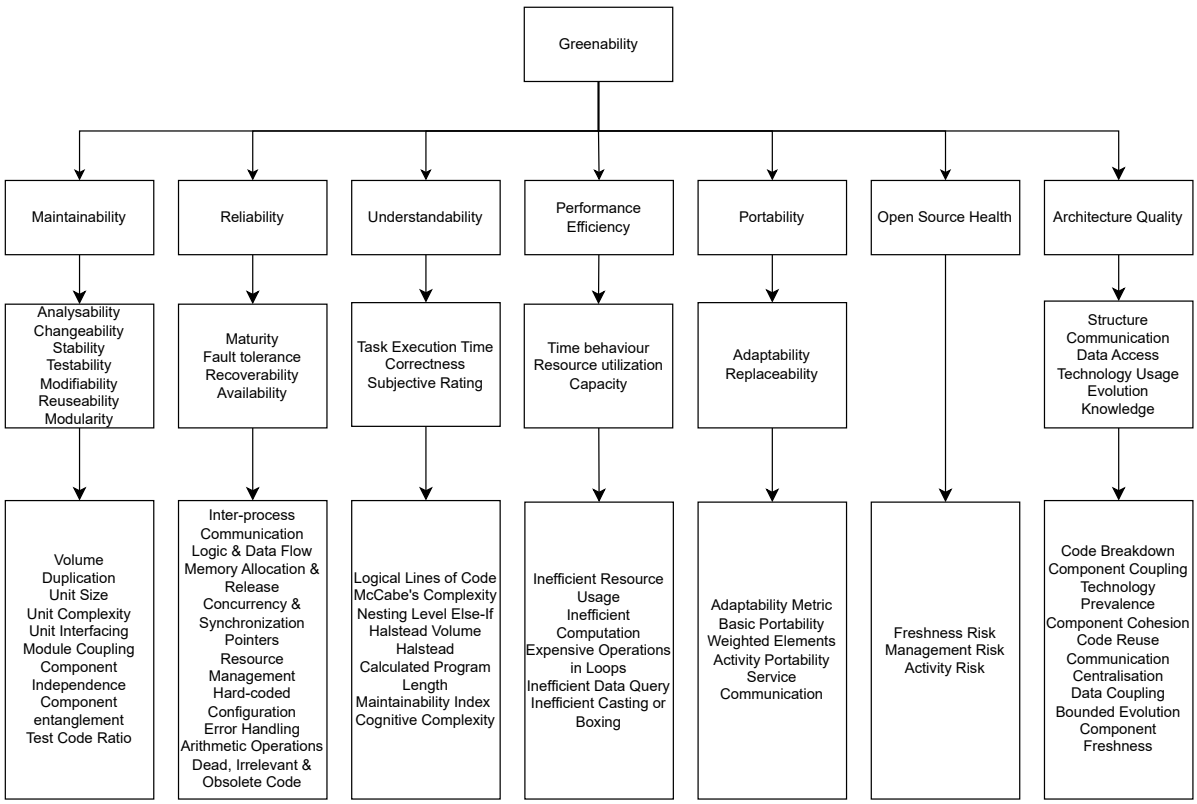


Figure 8.1: Greenability Properties, Sub-Characteristics and Metrics

The validity of the Greenability model was determined by comparing correlational analysis of refactoring effort (statically measured churn results) to developers' perceived effort. Additionally, a survey was conducted, which highlighted a strong correlation between Greenability metrics and properties, and the actual effort required for refactoring. This further validated the model's practical relevance. By combining these subjective perspectives with quantitative data, this project aimed to create a more comprehensive and validated model for measuring Greenability. This approach ensures that the Greenability metrics are both theoretically sound and practically relevant. Furthermore, the application of the Greenability model was shown to yield actionable insights and recommendations. For example, the model identified specific areas where code improvements could lead to significant energy savings. By developing a validated Greenability model, a valuable resource for the software engineering community has been provided to promote more sustainable software practices.

The answers to each of the four Research Questions are as follows:

RQ1: Which software quality properties determine software "Greenability"?

- Maintainability
- Reliability
- Understandability
- Performance Efficiency
- Portability
- Open Source Health
- Architecture Quality

On average, all refactorings are easier to implement if any Greenability Property is improved. This is true for both measured and perceived effort. Architecture Quality and Reliability are the Greenability Properties with the biggest measured impact on ease of refactorings.

RQ2: Which software quality metrics determine software "Greenability"?

This Research Question is most easily answered through a visual representation of the Greenability Model in Figure 8.1, by focusing specifically on the bottom row. Higher Greenability Metric scores, on average, reduce the effort required for all code refactorings. Volume has the greatest impact on the ease of code refactorings.

RQ3: How does the predictive accuracy of the model compare to developer's perceived refactoring effort?

Developers' perception of effort aligned accurately with measured effort results for Reliability, Freshness and Architecture Quality. Developers overestimate how much easier refactorings would be on systems with higher Maintainability scores. Maintainability is perceived by developers to have the highest impact on code refactoring effort, however, opinions are dispersed. Performance Efficiency and Freshness are perceived by software developers as having a lesser impact on refactoring effort compared to other Greenability Properties.

RQ4: How can the application of the model yield actionable, specific insights and recommendations to enhance software practices?

Two interesting findings concluded that firstly, larger software systems require more effort to refactor compared to smaller systems, however a lower percentage of the entire system is refactored compared to smaller systems. Secondly, refactorings that require more effort to implement, also refactor a higher percentage of the overall system's source code. By applying the Greenability Model, developers can identify specific software quality properties that are most in need of improvement to reduce refactoring effort. For instance, by targeting properties such as Architecture Quality or Reliability, developers can make strategic changes that significantly ease the refactoring process. The model's metrics provide quantifiable insights, allowing software developers to prioritise efforts on areas with the greatest impact on Software sustainability. By systematically addressing low-scoring properties and metrics, the model guides developers toward making incremental improvements that collectively enhance the overall greenability of the software, resulting in more efficient and sustainable development practices.

8.1 Future work

The static source code analysis tool, Sigrid, could have future work implemented by producing 5-star ratings based on benchmarks of the thousands of systems available to SIG, for all Quality Models as well as for all Performance and Reliability sub-characteristics.

The correlation between energy efficiency and maintainability is an interesting concept for further research, as knowing for certain that faster code correlates with more changeable code would provide a strong justification for including Performance metrics in Greenabilty calculations.

A significant contribution to future work could be to create the ideal dataset for experiments similar to those implemented in this project. This data set would have the following ideal characteristics:

- **System Selection:** Multiple software systems, preferably around 100, with a large range of software quality ratings, particularly in Volume and Maintainability scores, as well as significant test code coverage.

- **Refactoring Selection:** Refactor each system with a set of Green Software Patterns, particularly patterns which are the most used and produce structural changes to a code base.
- **Methodology:** Every refactoring should be implemented on every pattern. Most literature sources only implement some refactorings on some systems, resulting in fragmented data insufficient for aggregating results by system or refactoring. Consistent refactoring patterns across systems are essential for accurate correlation results and project validation. It is also important to implement the refactorings in every instance within the source code that is possible. The before and after versions of the source code should be made available to encourage replication and further future work from the software community.

With a dataset like this available, it would be possible to conduct comprehensive studies on the impact of various refactoring patterns across different software systems. This would enhance the understanding of Greenability and provide reliable data to support improvements in Software Sustainability.

Acknowledgements

I would like to express my deepest gratitude to those who have supported and contributed to this project.

My two supervisors at SIG, Pepijn van de Kamp and Chushu Gao, your constant feedback, engaging discussions, and specific input on the survey were significant driving forces throughout this research. Thank you for your dedication and involvement.

To my academic supervisor, Ana Oprescu, our regular meetings and feedback provided continuous inspiration and motivation. I am deeply appreciative of your guidance and encouragement.

Aspasia Koukouvela and Philipp Sommerhalter, your engaging discussions and brainstorming sessions greatly influenced the direction of this research. Your companionship and perspectives were essential in shaping the final outcome.

This project would not have been possible without the resources and environment provided by the University of Amsterdam (UvA) and the Software Improvement Group (SIG). I am grateful to SIG for providing a work environment, access to tools, and troubleshooting support for Sigrid. Special thanks to Marco di Biase for his guidance on the Open Source Health Sigrid functionalities, and the encouragement from fellow interns, Floris van Leeuwen and Tomás Candeias. I would also like to extend my appreciation to all my colleagues at SIG for the opportunity to learn from them, and for allowing me to present my work to experts in the field. Their technical support was crucial in navigating the complexities of the project.

Lastly, I extend my gratitude to all the respondents of my survey, whose participation and input were critical in validating the research findings, as well as my family and friends. Thank you all for your support and contributions.

Bibliography

[1] C. Calero, M. Á. Moraga, and M. Piattini, Eds., *Software Sustainability*, Cham: Springer International Publishing, 2021, ISBN: 978-3-030-69970-3. DOI: 10.1007/978-3-030-69970-3. [Online]. Available: https://link.springer.com/10.1007/978-3-030-69970-3 (visited on 01/10/2024).

[2] A. Noureddine, R. Rouvoy, and L. Seinturier, "A review of energy measurement approaches," *SIGOPS Oper. Syst. Rev.*, vol. 47, no. 3, pp. 42–49, Nov. 2013, ISSN: 0163-5980. DOI: 10.1145/2553070.2553077. [Online]. Available: https://doi.org/10.1145/2553070.2553077.

[3] S. Naumann, M. Dick, E. Kern, and T. Johann, "The GREENSOFT model: A reference model for green and sustainable software and its engineering," *Sustainable Computing: Informatics and Systems*, vol. 1, no. 4, pp. 294–304, Dec. 2011, ISSN: 22105379. DOI: 10.1016/j.suscom.2011.06.004. [Online]. Available: https://linkinghub.elsevier.com/retrieve/pii/S2210537911000473 (visited on 01/10/2024).

[4] G. Kalaitzoglou, M. Bruntink, and J. Visser, "A practical model for evaluating the energy efficiency of software applications:" *ICT for Sustainability 2014 (ICT4S-14)*, 2014. DOI: 10.2991/ict4s-14.2014.9. [Online]. Available: https://www.atlantis-press.com/article/13427 (visited on 01/10/2024).

[5] R. Baggen, J. P. Correia, K. Schill, and J. Visser, "Standardized code quality benchmarking for improving software maintainability," *Software Quality Journal*, vol. 20, no. 2, pp. 287–307, 2012. DOI: 10.1007/s11219-011-9144-9. [Online]. Available: https://doi.org/10.1007/s11219-011-9144-9.

[6] M. Longo, A. Rodriguez, C. Mateos, and A. Zunino, "Reducing energy usage in resource-intensive java-based scientific applications via micro-benchmark based code refactorings," *Computer Science and Information Systems*, vol. 16, pp. 9–9, Jan. 2019. DOI: 10.2298/CSIS180608009L.

[7] L. Koedijk and A. Oprescu, "Finding significant differences in the energy consumption when comparing programming languages and programs," in *2022 International Conference on ICT for Sustainability (ICT4S)*, Plovdiv, Bulgaria: IEEE, Jun. 2022, pp. 1–12, ISBN: 978-1-66548-286-8. DOI: 10.1109/ICT4S55073.2022.00012. [Online]. Available: https://ieeexplore.ieee.org/document/9830107/ (visited on 01/23/2024).

[8] C. Sahin, L. L. Pollock, and J. A. Clause, "How do code refactorings affect energy usage?" In *International Symposium on Empirical Software Engineering and Measurement*, 2014. [Online]. Available: https://api.semanticscholar.org/CorpusID:14237840.

[9] M. Dick and S. Naumann, "Enhancing software sustainability: A case study," *IT Professional*, vol. 12, no. 1, pp. 32–39, 2010.

[10] ISO, "Iso/iec 25010 standard: Software engineering – software product quality requirements and evaluation (square) – guide to square," International Organization for Standardization, Tech. Rep., 2005. [Online]. Available: https://iso25000.com/index.php/en/iso-25000-standards/iso-25010.

[11] D. Feitosa, L. Cruz, R. Abreu, J. P. Fernandes, M. Couto, and J. Saraiva, "Patterns and energy consumption: Design, implementation, studies, and stories," in *Software Sustainability*, C. Calero, M. Á. Moraga, and M. Piattini, Eds. Cham: Springer International Publishing, 2021, pp. 89–121, ISBN: 978-3-030-69970-3. DOI: 10.1007/978-3-030-69970-3_5. [Online]. Available: https://doi.org/10.1007/978-3-030-69970-3_5.

[12] J. Britton, *What is iso 25010?* May 2021. [Online]. Available: https://www.perforce.com/blog/qac/what-is-iso-25010.

[13] ISO, *"iso/iec 9126-1: Software engineering - product quality - part 1: Quality model"*, Geneva, Switzerland, 2001. [Online]. Available: https://standards.iteh.ai/catalog/standards/sist/d4ab62c2-7a1f-4586-b33d-25bcf8cf19c1/iso-iec-9126-1-2001.

[14] cisq. "Software quality standards – iso 5055." (), [Online]. Available: https://www.it-cisq.org/standards/code-quality-standards/.

[15] L. Lavazza, S. Morasca, and M. Gatto, "An empirical study on software understandability and its dependence on code characteristics," eng, *Empirical software engineering : an international journal*, vol. 28, no. 6, pp. 155–, 2023, ISSN: 1382-3256.

[16] C. Calero, B. Manuel, and D. Leticia, "Quality in use and software greenability," *CEUR Workshop Proceedings*, vol. 1216, p. 9, 2014, ISSN: 16130073. [Online]. Available: https://ceur-ws.org/Vol-1216/paper6.pdf (visited on 03/02/2024).

[17] SIG. [Online]. Available: https://docs.sigrid-says.com/.

[18] B. Kitchenham and S. Charters, "Guidelines for performing systematic literature reviews in software engineering," EBSE Technical Report EBSE-2007-01, Tech. Rep., 2007. [Online]. Available: https://www.elsevier.com/__data/promis_misc/525444systematicreviewsguide.pdf.

[19] R. Vis, D. Bijlsma, and H. Xu, *Sig/tüvit evaluation criteria trusted product maintainability*, version 15, Aug. 2023.

[20] D. Bijlsma and M. Olivari, *Sig architecture quality model guidance for producers*, 2023.

[21] R. van der Veer, *Sig evaluation criteria security: Guidance for producers*, version 3, 2023.

[22] "How to accurately measure the energy consumption of application software," Green Software Foundation. (Jun. 5, 2023), [Online]. Available: https://greensoftware.foundation/articles/how-to-accurately-measure-the-energy-consumption-of-application-software (visited on 01/10/2024).

[23] "Green software patterns," Green Software Foundation. (), [Online]. Available: https://patterns.greensoftware.foundation// (visited on 01/10/2024).

[24] *Green-software-foundation/patterns*, original-date: 2022-06-27T22:01:30Z, Dec. 16, 2023. [Online]. Available: https://github.com/Green-Software-Foundation/patterns (visited on 01/10/2024).

[25] W. Oliveira, H. Matalonga, G. Pinto, F. Castor, and J. P. Fernandes, "Small changes, big impacts: Leveraging diversity to improve energy efficiency," in *Software Sustainability*, C. Calero, M. Á. Moraga, and M. Piattini, Eds., Cham: Springer International Publishing, 2021, pp. 123–152, ISBN: 978-3-030-69969-7. DOI: 10.1007/978-3-030-69970-3_6. [Online]. Available: https://link.springer.com/10.1007/978-3-030-69970-3_6 (visited on 01/24/2024).

[26] A. Guldner, E. Kern, S. Kreten, and S. Naumann, "Criteria for sustainable software products: Analyzing software, informing users, and politics," in *Software Sustainability*, C. Calero, M. Á. Moraga, and M. Piattini, Eds., Cham: Springer International Publishing, 2021, pp. 17–42, ISBN: 978-3-030-69969-7. DOI: 10.1007/978-3-030-69970-3_2. [Online]. Available: https://link.springer.com/10.1007/978-3-030-69970-3_2 (visited on 01/24/2024).

[27] I. Heitlager, T. Kuipers, and J. Visser, "A practical model for measuring maintainability," in *Quality of Information and Communications Technology, 6th International Conference on the Quality of Information and Communications Technology, QUATIC 2007, Lisbon, Portugal, September 12-14, 2007, Proceedings*, 2007, pp. 30–39. DOI: 10.1109/QUATIC.2007.8. [Online]. Available: https://doi.org/10.1109/QUATIC.2007.8.

[28] F. Albertao, J. Xiao, C. Tian, Y. Lu, K. Q. Zhang, and C. Liu, "Measuring the sustainability performance of software projects," in *2010 IEEE 7th International Conference on E-Business Engineering*, Shanghai, China: IEEE, Nov. 2010, pp. 369–373, ISBN: 978-1-4244-8386-0. DOI: 10.1109/ICEBE.2010.26. [Online]. Available: http://ieeexplore.ieee.org/document/5704342/ (visited on 01/24/2024).

[29] F. Albertao. "Sustainable software engineering," Scribd. (), [Online]. Available: https://www.scribd.com/document/5507536/Sustainable-Software-Engineering.

[30] J. Nielsen, *Usability Engineering*. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 1994, ISBN: 9780080520292.

[31] J. Lenhard, "Improving process portability through metrics and continuous inspection," in *Advances in Intelligent Process-Aware Information Systems*, ser. Intelligent Systems Reference Library, G. G. Manfred Reichert Roy Oberhauser, Ed., vol. 123, Springer, 2017, pp. 193–223, ISBN: 978-3-319-52179-4. DOI: 10.1007/978-3-319-52181-7_8. [Online]. Available: https://urn.kb.se/resolve?urn=urn:nbn:se:kau:diva-54034.

[32] J. Lenhard and G. Wirtz, "Measuring the portability of executable service-oriented processes," eng, in *2013 17th IEEE International Enterprise Distributed Object Computing Conference*, IEEE, 2013, pp. 117–126, ISBN: 0769550819.

[33] A. Beer, M. Felderer, T. Lorey, and S. Mohacsi, "Aspects of sustainable test processes," in *2021 IEEE/ACM International Workshop on Body of Knowledge for Software Sustainability (BoKSS)*, 2021, pp. 9–10. DOI: 10.1109/BoKSS52540.2021.00012.

[34] P. Leitner and C.-P. Bezemer, "An exploratory study of the state of practice of performance testing in java-based open source projects," in *Proceedings of the 8th ACM/SPEC on International Conference on Performance Engineering*, L'Aquila Italy: ACM, Apr. 17, 2017, pp. 373–384, ISBN: 978-1-4503-4404-3. DOI: 10.1145/3030207.3030213. [Online]. Available: https://dl.acm.org/doi/10.1145/3030207.3030213 (visited on 02/07/2024).

[35] M. Rosado de Souza, R. Haines, M. Vigo, and C. Jay, "What makes research software sustainable? an interview study with research software engineers," in *2019 IEEE/ACM 12th International Workshop on Cooperative and Human Aspects of Software Engineering (CHASE)*, 2019, pp. 135–138. DOI: 10.1109/CHASE.2019.00039.

[36] F. D. Ajami S Woodbridge Y, "Syntax, predicates, idioms - what really affects code complexity.," *Empir Softw Eng*, pp. 287–328, 2019. DOI: https://doi.org/10.1007/s10664-018-9628-3.

[37] P. B. Börstler J, "The role of method chains and comments in software readability and comprehension - an experiment," *IEEE Trans Software Eng*, pp. 886–898, 2016. DOI: https://doi.org/10.1109/TSE.2016.2527791.

[38] J. Dolado, M. Harman, M. Otero, and L. Hu, "An empirical investigation of the influence of a type of side effects on program comprehension," *IEEE Trans Software Eng*, vol. 29, pp. 665–670, 2003. DOI: https://doi.org/10.1109/TSE.2003.1214329.

[39] R. Buse and W. Weimer, "Learning a metric for code readability," *IEEE Trans Software Eng*, vol. 26, pp. 546–558, 2010. DOI: https://doi.org/10.1109/TSE.2009.70.

[40] F. B, S. T, and W. W, "Decoding the representation of code in the brain: An fmri study of code review and expertise," *2017 IEEE/ACM 39th international conference on software engineering*, pp. 175–186, 2017.

[41] D. Fucci, D. Girardi, N. Novielli, L. Quaranta, and F. Lanubile, "A replication study on code comprehension and expertise using lightweight biometric sensors," *2019 IEEE/ACM 27th international conference on program comprehension (ICPC)*, pp. 311–322, 2019.

[42] A. Guldner *et al.*, "Development and evaluation of a reference measurement model for assessing the resource and energy efficiency of software products and components—green software measurement model (gsmm)," eng, *Future generation computer systems*, vol. 155, pp. 402–418, 2024, ISSN: 0167-739X.

[43] T. Enocsson and R. Kullang. "Cost-efficiency unleashed: The power of modularity in software development." (), [Online]. Available: https://www.modularmanagement.com/blog/cost-efficiency-unleashed-the-power-of-modularity-in-software-development.

[44] A. Loja and T. Maita, "Comparative evaluation of performance efficiency in terms of temporal behavior and resource utilization, according to the iso/iec 25,010 model, in a web application developed with angular, react.js, and vue.js," in *Emerging Research in Intelligent Systems*, G. F. Olmedo Cifuentes, D. G. Arcos Avilés, and H. V. Lara Padilla, Eds., Cham: Springer Nature Switzerland, 2024, pp. 293–308.

[45] Sigrid. "Sig open source health quality model 2024: Guidance for producers." (), [Online]. Available: https://docs.sigrid-says.com/reference/quality-model-documents/open-source-health.html.

[46] A. Wiggins. "The twelve factor app." (), [Online]. Available: https://12factor.net/.

[47] "Heroku." (), [Online]. Available: https://www.heroku.com/.

[48] M. Halstead, "Elements of software science," *Elsevier North-Holland*, 1977.

[49] W. KD, O. PW, and A. GG, "Development and application of an automated source code maintainability index.," *J Softw Maint Res Pract*, vol. 9, no. 3, pp. 127–159, 1997.

[50] C. GA, *Cognitive complexity - a new way of measuring understandability*. 2018. [Online]. Available: https://www.%20sonarsource.com/docs/CognitiveComplexity.pdf.

[51] L. Cruz and R. Abreu, "Performance-based guidelines for energy efficient mobile applications," in *2017 IEEE/ACM 4th International Conference on Mobile Software Engineering and Systems (MOBILESoft)*, 2017, pp. 46–57. DOI: 10.1109/MOBILESoft.2017.19.

[52] T. Vartziotis *et al.*, *Learn to code sustainably: An empirical study on llm-based green code generation*, 2024. arXiv: 2403.03344 [cs.SE]. [Online]. Available: https://arxiv.org/abs/2403.03344.

[53] L. Cruz, R. Abreu, and J.-N. Rouvignac, "Leafactor: Improving energy efficiency of android apps via automatic refactoring," in *2017 IEEE/ACM 4th International Conference on Mobile Software Engineering and Systems (MOBILESoft)*, 2017, pp. 205–206. DOI: 10.1109/MOBILESoft.2017.21.

[54] A. Hindle, "Green mining: A methodology of relating software change and configuration to power consumption," in *Empirical Software Engineering*, vol. 20, 2015, pp. 374–409.

[55] W. e. a. Oliveira, "Improving energy-efficiency by recommending java collections," in *Empir Software Eng*, vol. 26, 2021. [Online]. Available: https://doi-org.proxy.uba.uva.nl/10.1007/s10664-021-09950-y.

[56] F. Palomba, D. Di Nucci, A. Panichella, A. Zaidman, and A. De Lucia, "On the impact of code smells on the energy consumption of mobile applications," *Information and Software Technology*, vol. 105, pp. 43–55, 2019, ISSN: 0950-5849. DOI: https://doi.org/10.1016/j.infsof.2018.08.004. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0950584918301678.

[57] Z. Ournani, R. Rouvoy, and P. Rust, "Evaluating the energy consumption of java i/o apis," in *2021 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, 2021, pp. 1–11. DOI: 10.1109/ICSME52107.2021.00007.

[58] Z. Ournani, P. R. R Rouvoy, and J. Penhoat, "Tales from the code 1: The effective impact of code refactorings on software energy consumption," in *HAL open science*, 2021.

[59] D. Mclntire, K. Ho, B. Yip, A. Singh, W. Wu, and W. Kaiser, "The low power energy aware processing (leap) embedded networked sensor system," in *2006 5th International Conference on Information Processing in Sensor Networks*, 2006, pp. 449–457.

[60] A. Bertolino and L. Strigini, "On the use of testability measures for dependability assessment," *IEEE Transactions on Software Engineering*, vol. 22, no. 2, pp. 97–108, 1996. DOI: 10.1109/32.485220.

[61] M. Alenezi, "Investigating software testability and test cases effectiveness," *ArXiv*, vol. abs/2201.10090, 2022. [Online]. Available: https://api.semanticscholar.org/CorpusID:246275940.

[62] G. J. Myers, C. Sandler, and T. Badgett, *The art of software testing*, 3rd ed. Hoboken, N.J: John Wiley & Sons, 2012, 240 pp., OCLC: ocn728656684, ISBN: 978-1-118-03196-4.

[63] R. Verdecchia, P. Lago, C. Ebert, and C. de Vries, "Green IT and green software," *IEEE Software*, vol. 38, no. 6, pp. 7–15, Nov. 2021, Conference Name: IEEE Software, ISSN: 1937-4194. DOI: 10.1109/MS.2021.3102254. [Online]. Available: https://ieeexplore.ieee.org/document/9585139 (visited on 01/29/2024).

[64] T. E. Team, "Static analysis vs dynamic analysis in software testing," *TopTut*, 2023. [Online]. Available: https://www.toptut.com/2023/01/05/static-analysis-vs-dynamic-analysis-in-software-testing/.

[65] V. Cortellessa, A. Di Marco, and P. Inverardi, *Model-Based Software Performance Analysis*. Springer Berlin, Heidelberg, 2011. DOI: https://doi.org/10.1007/978-3-642-13621-4.

[66] "Running average power limit energy reporting / cve-2020-8694 , cve-2020-8695 / intel-sa-00389," Intel. (Feb. 2022), [Online]. Available: https://www.intel.com/content/www/us/en/developer/articles/technical/software-security-guidance/advisory-guidance/running-average-power-limit-energy-reporting.html (visited on 01/24/2024).

[67] A. Kansal, F. Zhao, J. Liu, N. Kothari, and A. A. Bhattacharya, "Virtual machine power metering and provisioning," in *ACM Symposium on Cloud Computing*, 2010. [Online]. Available: https://api.semanticscholar.org/CorpusID:12234071.

[68] "Codecarbon," https://github.com/mlco2/codecarbon. [Online]. Available: https://codecarbon.io/.

[69] J. Arnoldus, J. Gresnigt, K. Grosskop, and J. Visser, "Energy-efficiency indicators for e-services," in *Proceedings of the 2nd International Workshop on Green and Sustainable Software*, ser. GREENS '13, San Francisco, California: IEEE Press, 2013, pp. 24–29, ISBN: 9781467362672.

[70] D. Wong and M. Annavaram, "Knightshift: Scaling the energy proportionality wall through server-level heterogeneity," in *2012 45th Annual IEEE/ACM International Symposium on Microarchitecture*, 2012, pp. 119–130. DOI: 10.1109/MICRO.2012.20.

[71] T. A. Ghaleb, "Software energy measurement at different levels of granularity," in *2019 International Conference on Computer and Information Sciences (ICCIS)*, Sakaka, Saudi Arabia: IEEE, Apr. 2019, pp. 1–6, ISBN: 978-1-5386-8125-1. DOI: 10.1109/ICCISci.2019.8716456. [Online]. Available: https://ieeexplore.ieee.org/document/8716456/ (visited on 01/10/2024).

[72] R. Pereira *et al.*, "Energy efficiency across programming languages: How do energy, time, and memory relate?" In *Proceedings of the 2017 ACM SIGPLAN International Conference on Software Language Engineering (SLE'17)*, New York, NY, USA: ACM, 2017, pp. 256–267, ISBN: 978-1-4503-5525-4. DOI: 10.1145/3136014.3136031.

Acronyms

API Application Programming Interface. 45
CSRD Corporate Sustainability Reporting Directive. 7, 12
CWE Common Weakness Enumeration. 13
ICT Information and Communication Technology. 7
ISO International Organization for Standardization. 8
LOC Lines of Code. 15
SIG Software Improvement Group. 8
SQuaRE Systems and software Quality Requirements and Evaluation. 13

Appendix A

Refactoring Examples

Convert Local Variable to Field

Simple Example

Before:

```
public class MyClass {
    public void myMethod() {
        int myLocalVariable = 10;
        System.out.println(myLocalVariable);
    }
}
```

After:

```
public class MyClass {
    private int myField; // Converted from local variable

    public void myMethod() {
        myField = 10;
        System.out.println(myField);
    }
}
```

Real Example

Before:

```
public class BasicDynaBean implements DynaBean, Serializable {
...
 protected DynaProperty getDynaProperty(String name) {

        DynaProperty descriptor = getDynaClass().getDynaProperty(name);
        if (descriptor == null) {
            throw new IllegalArgumentException
                    ("Invalid property name '" + name + "'");
        }
        return (descriptor);

    }
}
```

After:

```
public class BasicDynaBean implements DynaBean, Serializable {
    public DynaProperty descriptor; // Converted from local variable
...
    protected DynaProperty getDynaProperty(String name) {

        descriptor = getDynaClass().getDynaProperty(name);
        if (descriptor == null) {
            throw new IllegalArgumentException
                    ("Invalid property name '" + name + "'");
        }
```

```
        return (descriptor);

    }
}
```

Extract Local Variable

Simple Example

Before:

```
public class MyClass {
    public void myMethod() {
        int result = 2 * (3 + 5);
        System.out.println(result);
    }
}
```

After:

```
public class MyClass {
    public void myMethod() {
        int sum = 3 + 5; // Extracted local variable
        int result = 2 * sum;
        System.out.println(result);
    }
}
```

Real Example

Before:

```
File rootFile = rootEntry.getFile();
if (rootFile.exists()) {
    checkAndNotify(rootEntry, rootEntry.getChildren(), listFiles(rootFile));
} else if (rootEntry.isExists()) {
    checkAndNotify(rootEntry, rootEntry.getChildren(), FileUtils.EMPTY_FILE_ARRAY);
}
```

After:

```
File rootFile = rootEntry.getFile();
FileEntry[] children = rootEntry.getChildren(); // Extracted local variable
if (rootFile.exists()) {
    checkAndNotify(rootEntry, children, listFiles(rootFile));
} else if (rootEntry.isExists()) {
    checkAndNotify(rootEntry, children, FileUtils.EMPTY_FILE_ARRAY);
}
```

Extract Method

Simple Example

Before:

```
public class MyClass {
    public void myMethod() {
        int result = 2 * (3 + 5);
        System.out.println(result);
    }
}
```

After:

```
public class MyClass {
    public void myMethod() {
        int result = calculateResult(); // Extracted method
        System.out.println(result);
    }
```

```
    private int calculateResult() {
        return 2 * (3 + 5);
    }
}
```

Real Example

Before:

```
public boolean addAll(Collection coll) {
    boolean changed = false;
    Iterator i = coll.iterator();
    while (i.hasNext()) {
        boolean added = add(i.next());
        changed = changed || added;
    }
    return changed;
}
```

After:

```
public boolean addAll(Collection coll) {
    boolean changed = false;
    changed = extractMethod(coll, changed);
    return changed;
}

private boolean extractMethod(Collection coll, boolean changed) {
    Iterator i = coll.iterator();
    while (i.hasNext()) {
        boolean added = add(i.next());
        changed = changed || added;
    }
    return changed;
}
```

Introduce Indirection

Simple Example

Before:

```
public class MyClass {
    public void myMethod() {
        originalMethod();
    }

    public void originalMethod() {
        System.out.println("Original method");
    }
}
```

After:

```
public class MyClass {
    public void myMethod() {
        indirectMethod(); // Introduced indirection
    }

    public static void indirectMethod() {
        MyClass myClass = new MyClass();
        myClass.originalMethod();
    }

    public void originalMethod() {
        System.out.println("Original method");
    }
}
```

Real Example

```
public abstract class AbstractMapBag implements Bag {
    private transient Map map;
    ...
    public int getCount(Object object) { // Original getCount method
        MutableInteger count = (MutableInteger) map.get(object);
        if (count != null) {
            return count.value;
        }
        return 0;
    }

    boolean containsAll(Bag other) {
        boolean result = true;
        Iterator it = other.uniqueSet().iterator();
        while (it.hasNext()) {
            Object current = it.next();
            boolean contains = getCount(current) >= other.getCount(current); // Original line
            result = result && contains;
        }
        return result;
    }
    ...
}
```

After:

```
public abstract class AbstractMapBag implements Bag {
    public static int getCount(Bag bag, Object object) { // Added static method
        return bag.getCount(object);
    }

    private transient Map map;
    ...
    public int getCount(Object object) { // Unchanged getCount method
        MutableInteger count = (MutableInteger) map.get(object);
        if (count != null) {
            return count.value;
        }
        return 0;
    }

    boolean containsAll(Bag other) {
        boolean result = true;
        Iterator it = other.uniqueSet().iterator();
        while (it.hasNext()) {
            Object current = it.next();
            boolean contains = AbstractMapBag.getCount(this, current) >= AbstractMapBag.getCount(other,
                current); // Changed line to use static method
            result = result && contains;
        }
        return result;
    }
    ...
}
```

Inline Method

Simple Example

Before:

```
public class MyClass {
    public void myMethod() {
        printMessage();
    }

    public void printMessage() {
        System.out.println("Hello, World!");
    }
}
```

After:

```
public class MyClass {
    public void myMethod() {
        System.out.println("Hello, World!"); // Inlined method
    }
}
```

Real Example

Before:

```
protected void removeMapping(HashEntry entry, int hashIndex, HashEntry previous) {
    modCount++;
    removeEntry(entry, hashIndex, previous);
    size--;
    destroyEntry(entry);
}

protected void destroyEntry(HashEntry entry) {
    entry.next = null;
    entry.key = null;
    entry.value = null;
}
```

After:

```
protected void removeMapping(HashEntry entry, int hashIndex, HashEntry previous) {
    modCount++;
    removeEntry(entry, hashIndex, previous);
    size--;
    entry.next = null; // inlined method
    entry.key = null;
    entry.value = null;
}
```

Introduce Parameter Object

Simple Example

Before:

```
public class MyClass {
    public void myMethod(String firstName, String lastName) {
        System.out.println(firstName + " " + lastName);
    }
}
```

After:

```
public class MyClass {
    public void myMethod(Person person) { // Introduced parameter object
        System.out.println(person.getFirstName() + " " + person.getLastName());
    }
}

class Person {
    private String firstName;
    private String lastName;

    public Person(String firstName, String lastName) {
        this.firstName = firstName;
        this.lastName = lastName;
    }

    public String getFirstName() {
        return firstName;
    }

    public String getLastName() {
        return lastName;
```

```
    }
}
```

Real Example

Before:

```
public abstract class AbstractRealDistribution implements RealDistribution, Serializable {
    double x = UnivariateSolverUtils.solve(toSolve,lowerBound,upperBound,getSolverAbsoluteAccuracy());
...
}
public class UnivariateSolverUtils {
    public static double solve(UnivariateFunction function, double x0,
                                                             double x1,
                                                             double absoluteAccuracy) {
    if (function == null) {
        throw new NullArgumentException(LocalizedFormats.FUNCTION);
    }
    final UnivariateSolver solver = new BrentSolver(absoluteAccuracy);
    return solver.solve(Integer.MAX_VALUE, function, x0, x1);
    }
...
}
```

After:

```
public class SolveParameter {
    public UnivariateFunction function;
    public double x0;
    public double x1;
    public double absoluteAccuracy;

    public SolveParameter(UnivariateFunction function, double x0, double x1, double absoluteAccuracy) {
    this.function = function;
    this.x0 = x0;
    this.x1 = x1;
    this.absoluteAccuracy = absoluteAccuracy;
  }
}

public abstract class AbstractRealDistribution implements RealDistribution, Serializable {
    double x = UnivariateSolverUtils.solve(new SolveParameter(toSolve, lowerBound, upperBound,
         getSolverAbsoluteAccuracy()));
...
}

public class UnivariateSolverUtils {
    public static double solve(SolveParameter parameterObject) {
        if (parameterObject.function == null) {
            throw new NullArgumentException(LocalizedFormats.FUNCTION);
        }
        final UnivariateSolver solver = new BrentSolver(parameterObject.absoluteAccuracy);
        return solver.solve(Integer.MAX_VALUE, parameterObject.function, parameterObject.x0,
             parameterObject.x1);
    }
...
}
```

Appendix B

Survey Questions

Page 1: Introduction to Survey

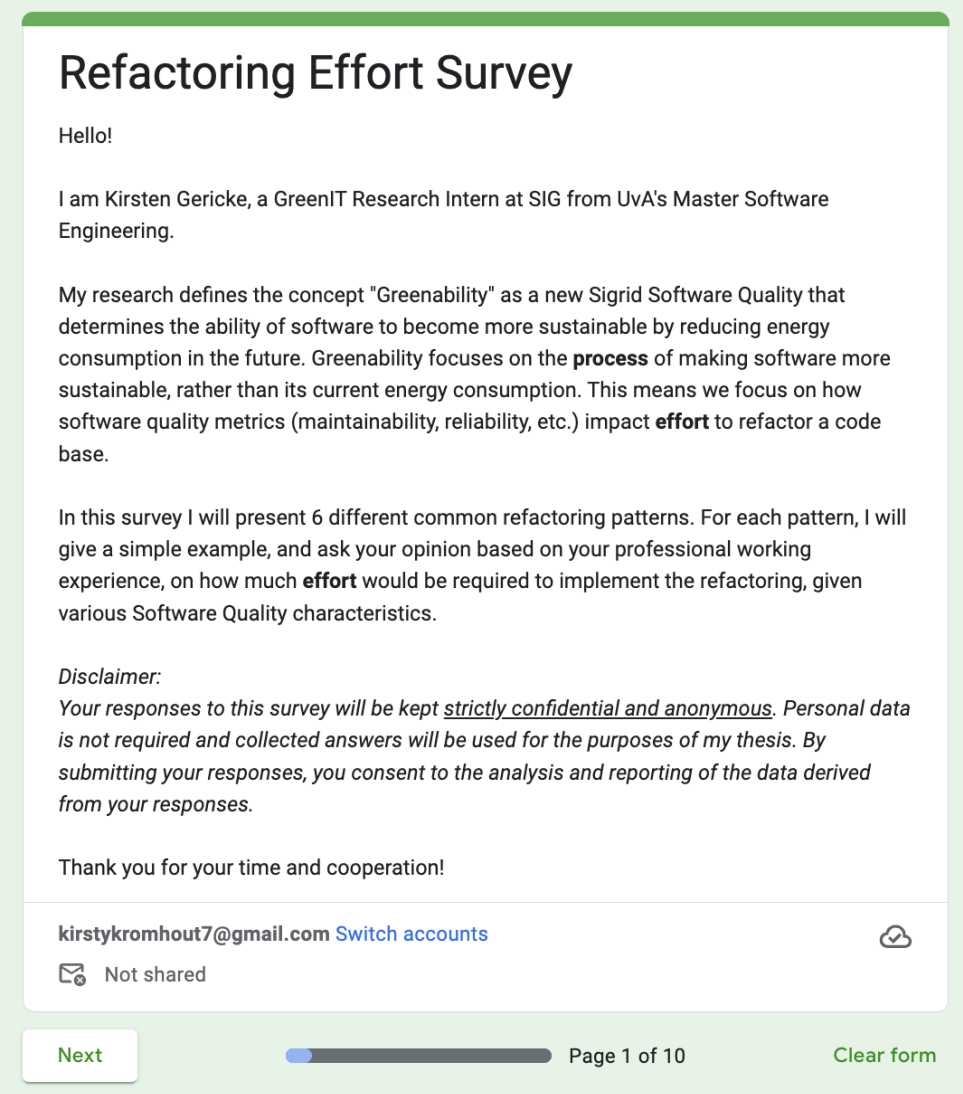


Figure B.1: Introduction to Survey

Page 2: Basic Information about Participant

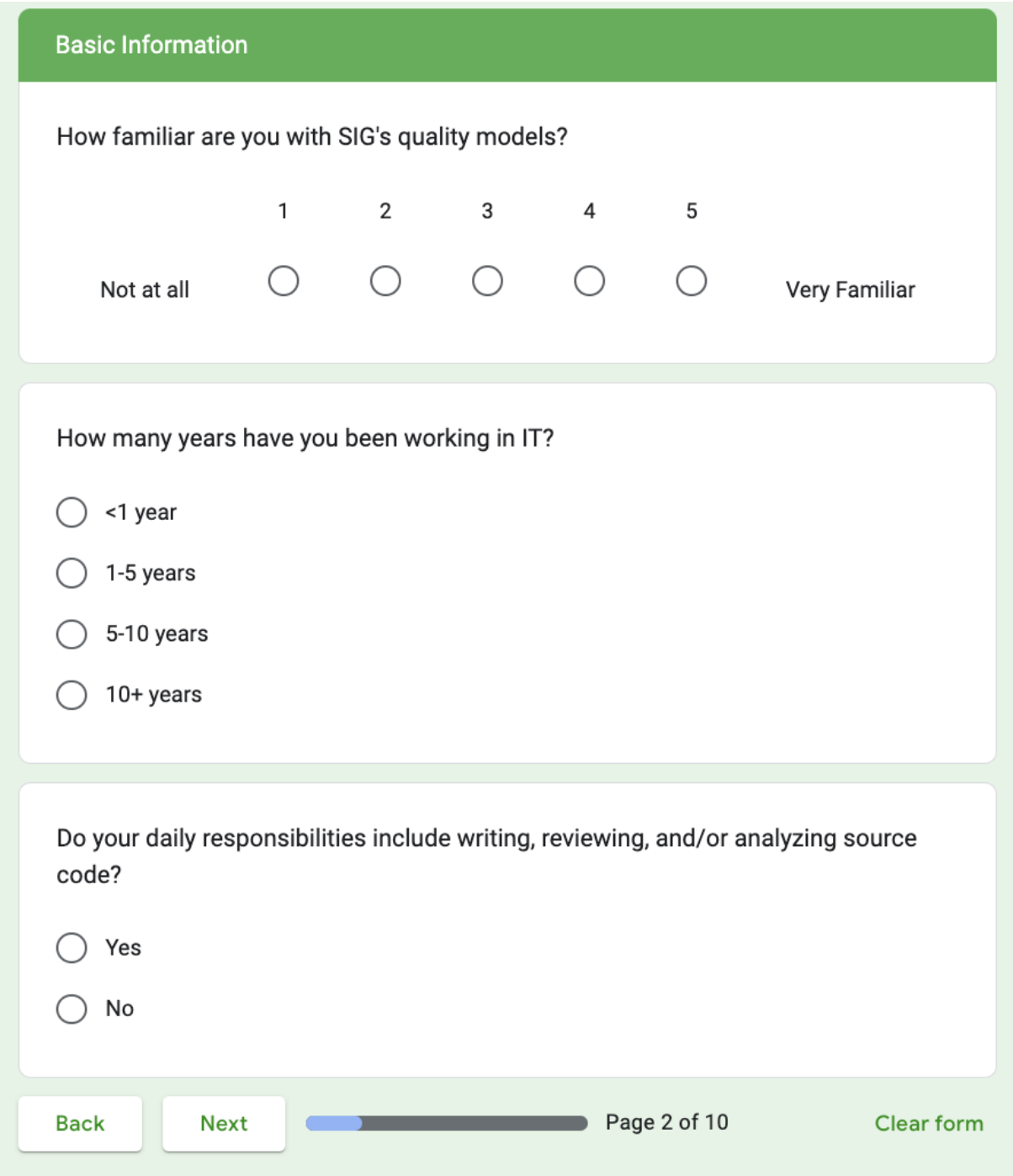


Figure B.2: Questions about participant

Page 3: Task Description

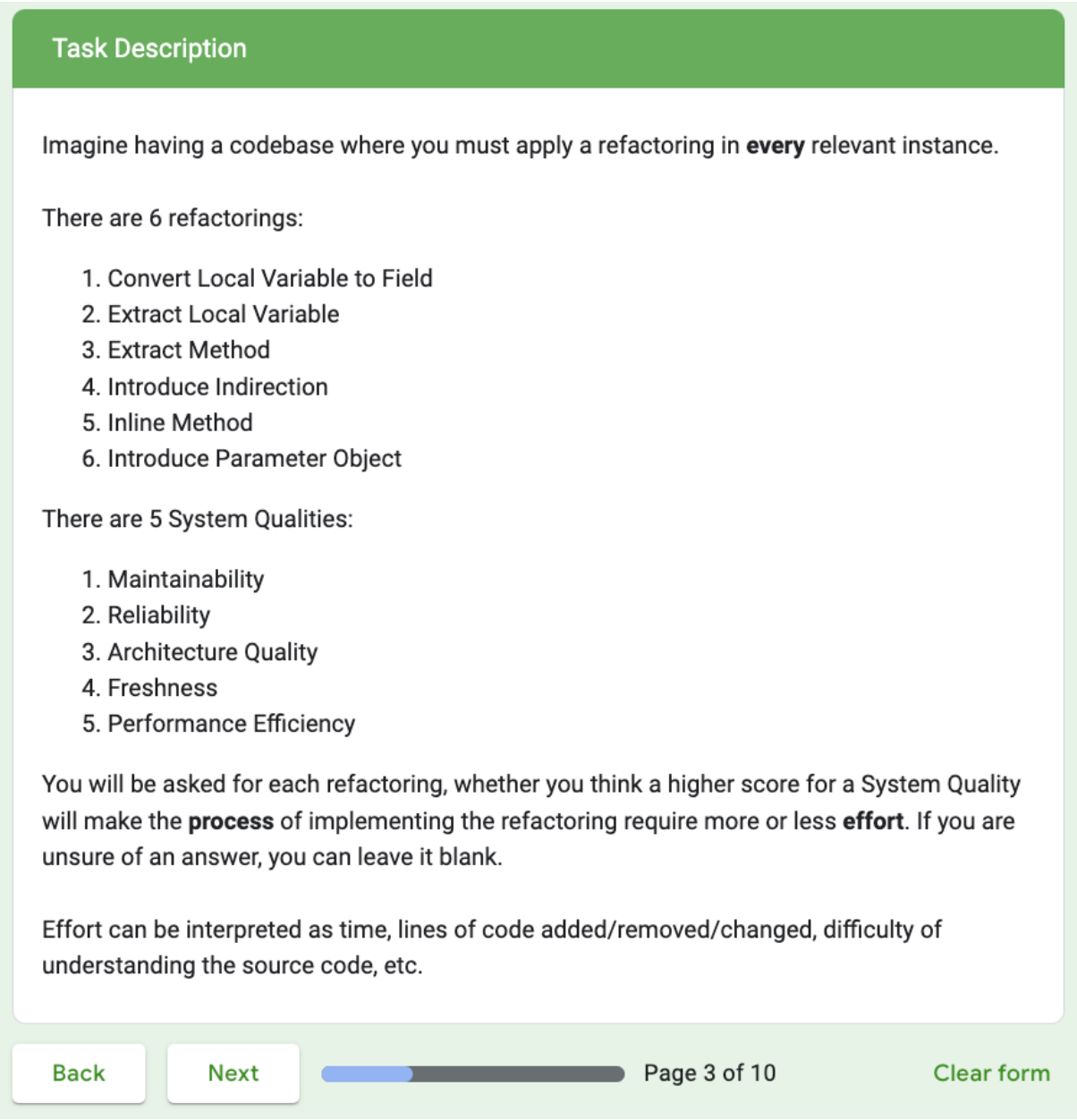


Figure B.3: Task Description

Page 4: Questions for Refactoring 1

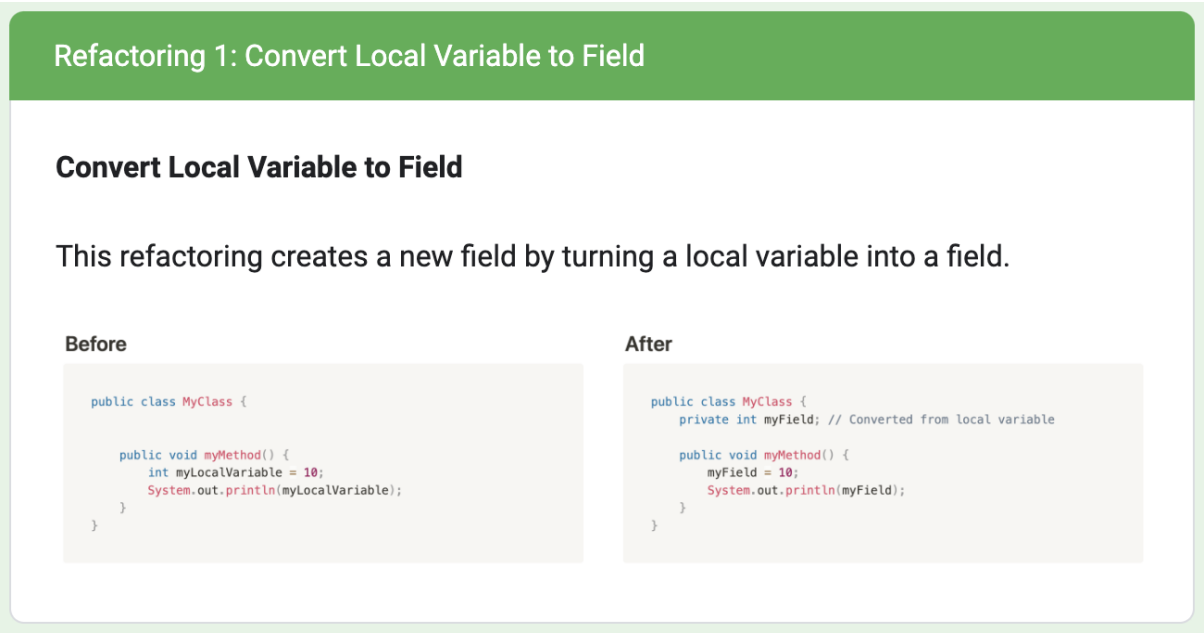


Figure B.4: Refactoring 1: Name, Description and Example

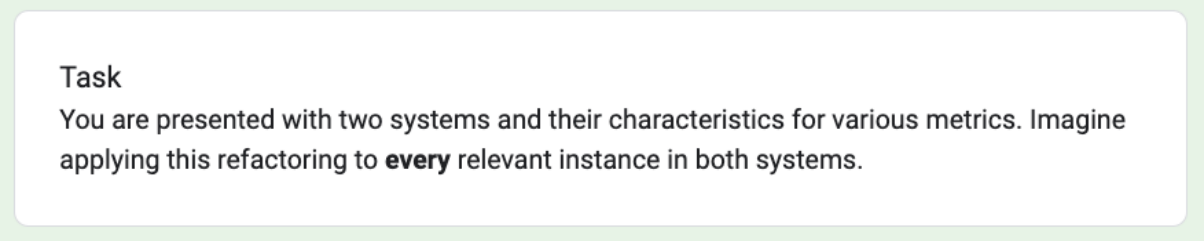


Figure B.5: Task clarification

Maintainability

Would implementing this refactoring in **System A** require more or less effort than in **System B**?

Metrics	System A	System B
Volume	10K LOC	50K LOC
Duplication	No duplicated lines	40% of the codebase is duplicated.
Unit Size	Average method size: 15 lines	Average method size: 50 lines
Unit Complexity	Average McCabe Complexity: 3	Average McCabe Complexity: 10
Module Coupling	Few parameters per method	Many parameters per method
Component Independence	Small interface	Many incoming calls to interface
Component Entanglement	Minimal component communication	High component communication

	1	2	3	4	5	
Less Effort	○	○	○	○	○	More Effort

Figure B.6: Maintainability Question and Example

Reliability

Would implementing this refactoring in **System A** require more or less effort than in **System B**?

Metrics	System A	System B
Inter-process Communication	Effective design patterns	Inefficient communication
Logic & Data Flow	Correct boolean logic	Contradicting if statements
Memory Allocation & Release	Properly allocated and freed memory	Memory leaks
Concurrency & Synchronization	Well-designed concurrency	Deadlocks, poor synchronization
Pointers	Proper pointer handling	Frequent null pointer exceptions
Resource Management	Scalable under load	Inefficient, poor scalability
Hard-coded Configuration	Flexible configurations used	Hard-coded configurations
Error Handling	Proper use of try-catch	Frequent unhandled exceptions
Arithmetic Operations	Correct precedence	Integer overflows, division by zero
Dead & Irrelevant Code	Clean and relevant codebase	Many dead code sections

	1	2	3	4	5	
Less Effort	○	○	○	○	○	More Effort

Figure B.7: Reliability Question and Example

Architecture Quality

Would implementing this refactoring in **System A** require more or less effort than in **System B**?

Metrics	System A	System B
Code Breakdown	Classes < 200 lines Methods < 20 lines	Classes > 1000 lines Methods > 100 lines
Component Coupling	Few component dependencies	Many component dependencies
Technology Prevalence	Uses Java 11, Spring Boot 3.0	Uses Java 6, outdated frameworks
Component Cohesion	Follows single responsibility principle	Many responsibilities per component
Code Reuse	No duplication across modules	Duplicate code across modules
Communication Centralization	80% internal centralized code	30% internal code, scattered
Data Coupling	Components use their own databases	Components share a single database
Bounded Evolution	10% co-evolution, changes localized	One change impacts entire code base
Knowledge Distribution	Knowledge spread across 5 people	Concentrated in 1-2 people
Component Freshness	Updated monthly with latest patches	Updated irregularly, every 2-3 years

	1	2	3	4	5	
Less Effort	○	○	○	○	○	More Effort

Figure B.8: Architecture Quality Question and Example

Freshness

Would implementing this refactoring in **System A** require more or less effort than in **System B**?

Metrics	System A	System B
Freshness Risk	Dependencies updated within the last 6 months	Dependencies not updated for over 2 years
Component Freshness	Components updated within the last month	Components not updated for over a year
Technology Prevalence	Uses the latest stable versions of technologies	Uses outdated versions of technologies

	1	2	3	4	5	
Less Effort	○	○	○	○	○	More Effort

Figure B.9: Freshness Question and Example

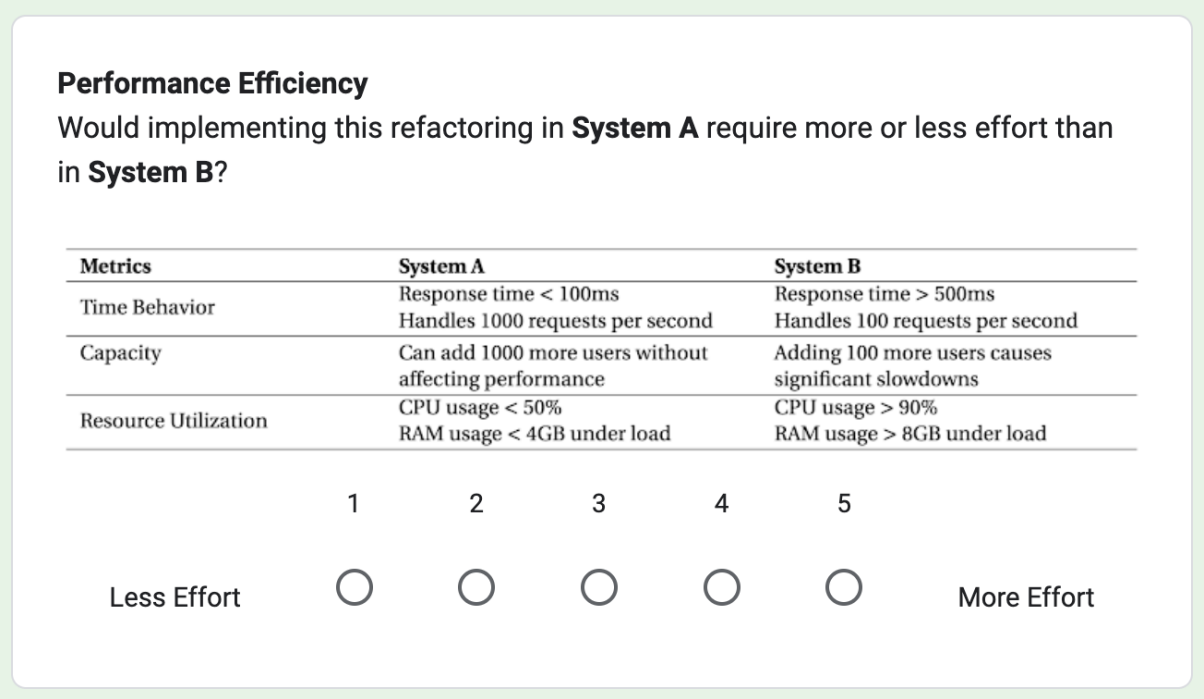


Figure B.10: Performance Efficiency Question and Example

Page 5: Questions for Refactoring 2

From this point on, the property questions and examples questions were identical for all refactorings, and are not repeated to reduce unnecessary space in this report being used.

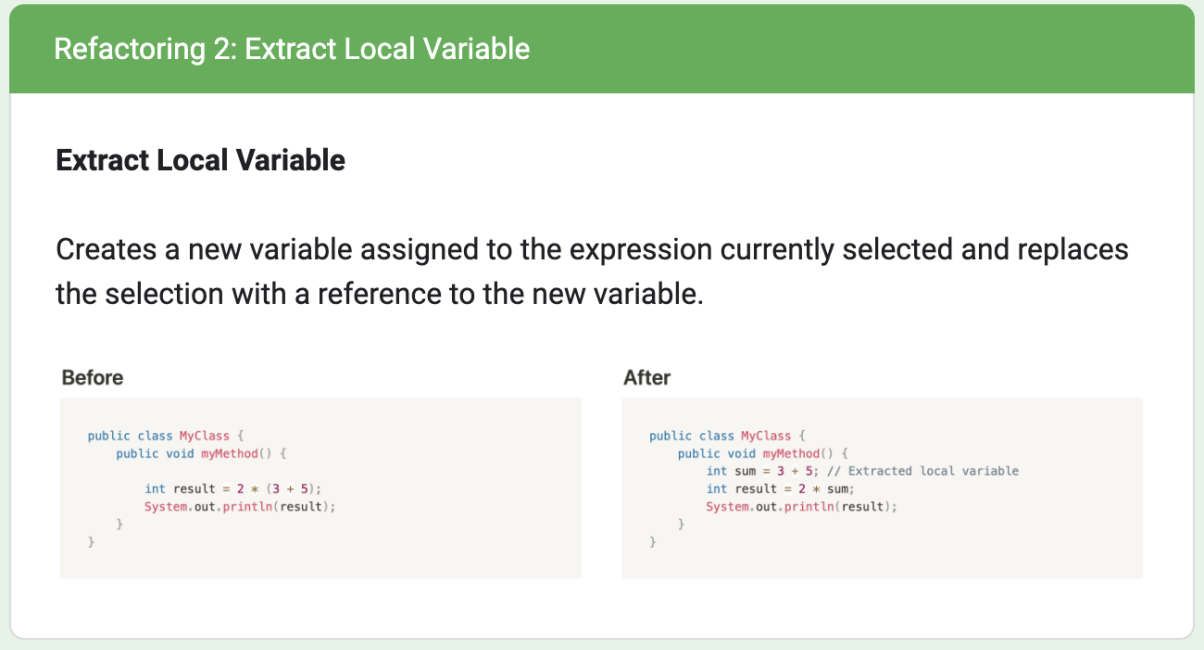


Figure B.11: Refactoring 2: Name, Description and Example

Page 6: Questions for Refactoring 3

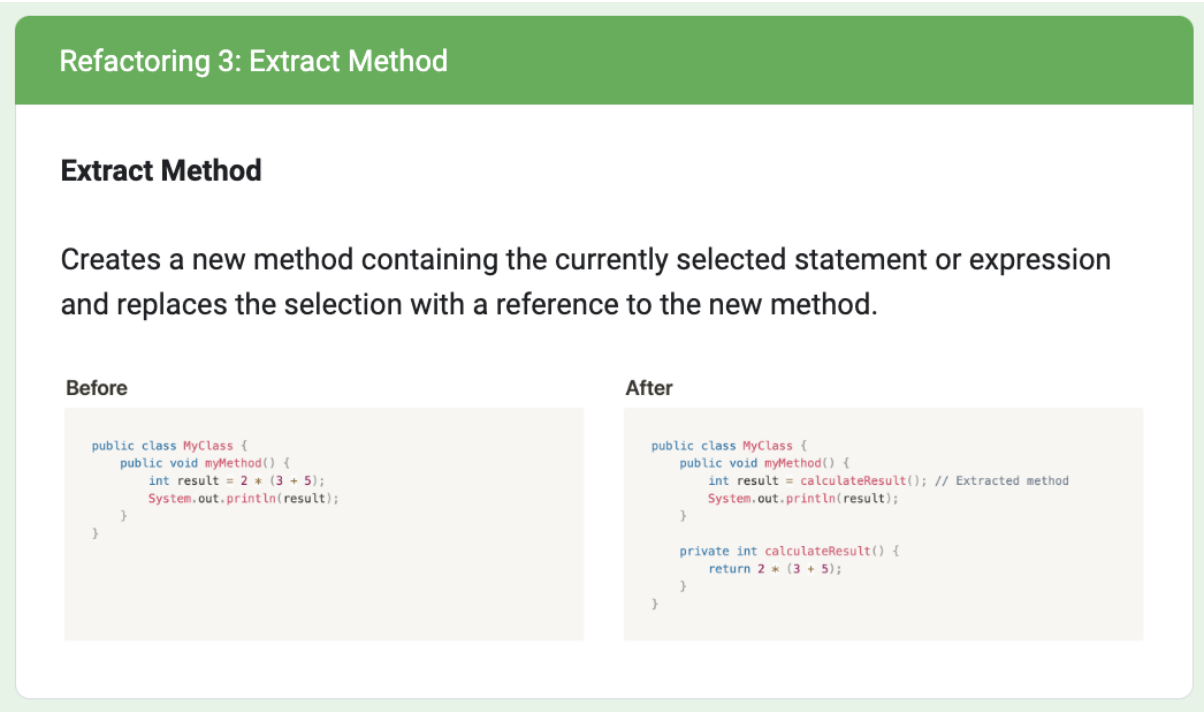


Figure B.12: Refactoring 3: Name, Description and Example

Page 7: Questions for Refactoring 4

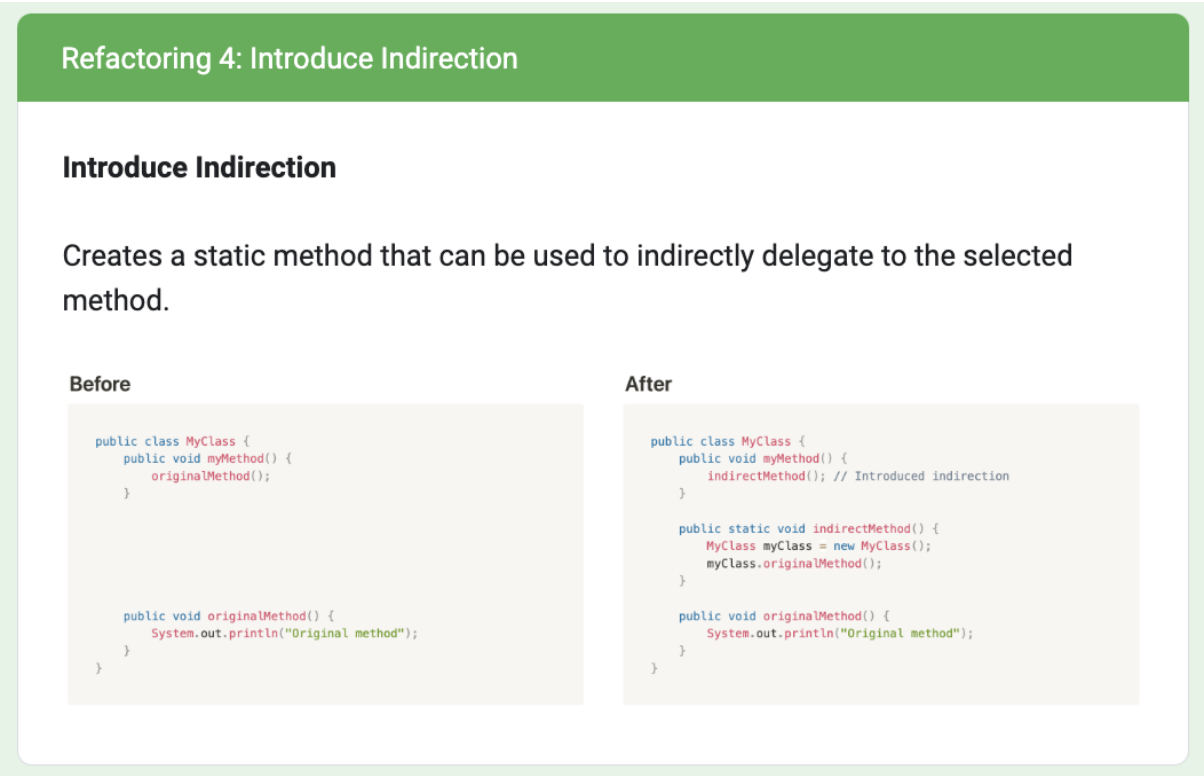


Figure B.13: Refactoring 4: Name, Description and Example

Page 8: Questions for Refactoring 5

Figure B.14: Refactoring 5: Name, Description and Example

Page 9: Questions for Refactoring 6

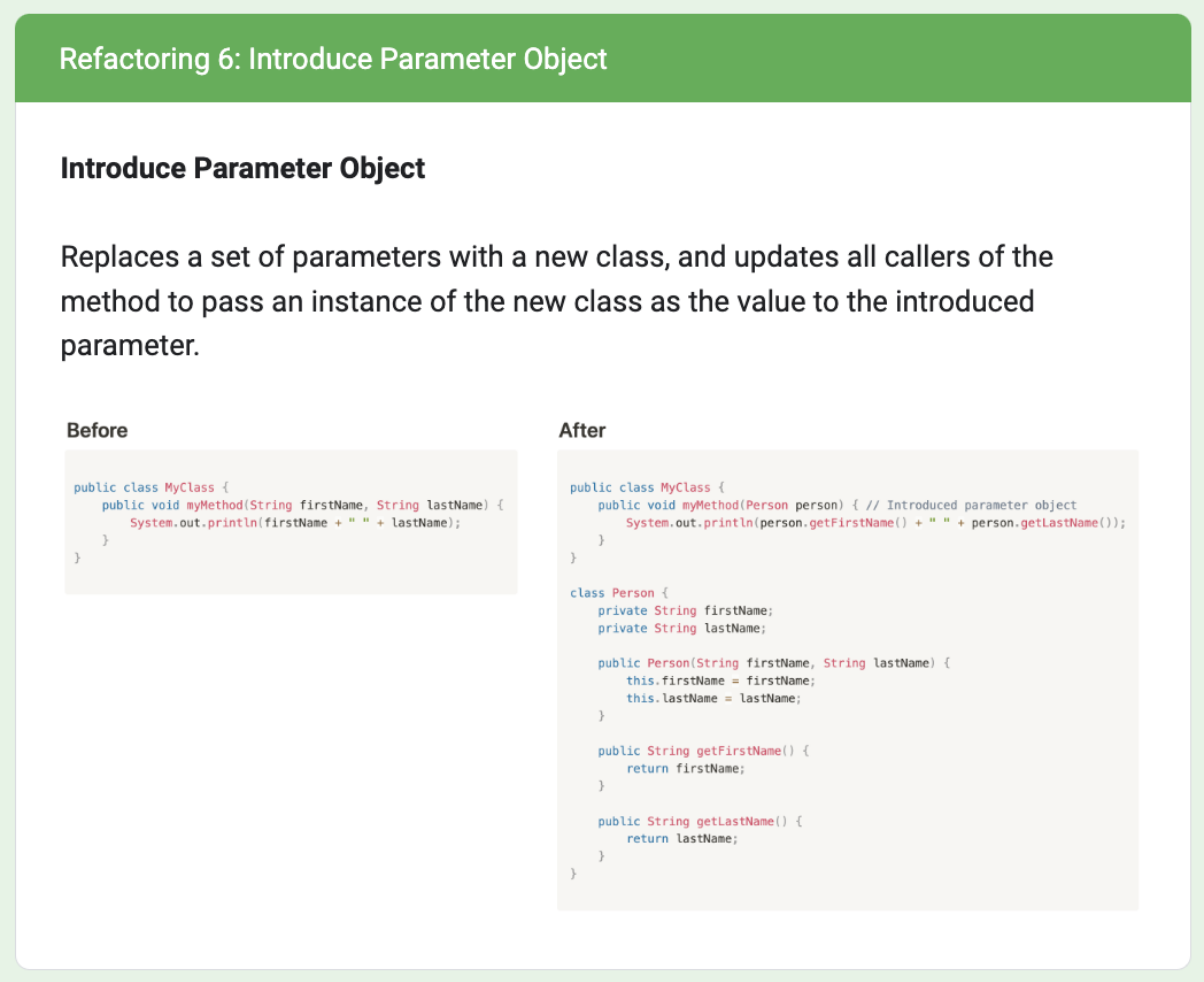


Figure B.15: Refactoring 6: Name, Description and Example

Page 10: Thank you and Additional Questions

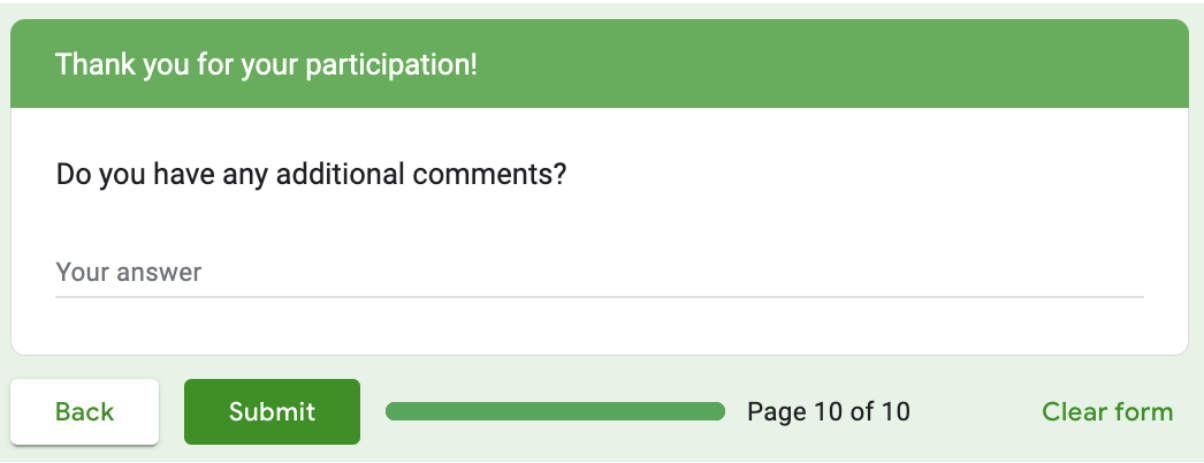
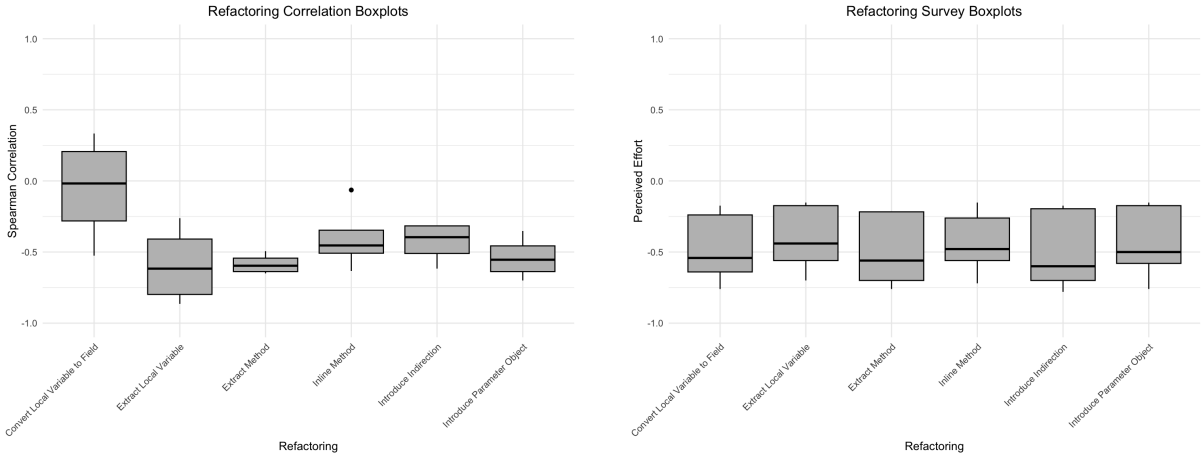


Figure B.16: Final Thanks and Additional Questions

Appendix C

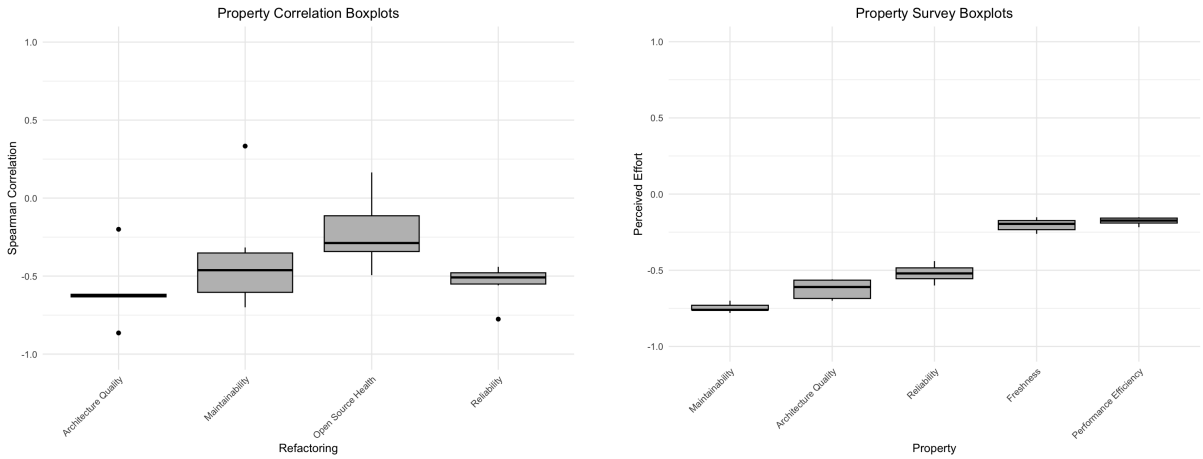
Box Plots



(a) Measured Effort **(b) Perceived Effort**

Figure C.1: Refactoring Results Box Plots

These survey results have been normalised from a range of 1 (less effort) to 5 (more effort), to a range of -1 to 1. This conversion is done so these results can be compared to the Spearman Correlation results, which is also on a scale from -1 (negative correlation) to 1 (positive correlation).



(a) Measured Effort **(b) Perceived Effort**

Figure C.2: Property Results Box Plots