HAW HAMBURG

BACHELORTHESIS
Nina Cathreen Godenrath

Fallstudienanalyse: Energieeffizienzsteigerung durch Refactoring von Java-Anwendungen

FAKULTÄT TECHNIK UND INFORMATIK
Department Informatik

Faculty of Computer Science and Engineering
Department Computer Science

HOCHSCHULE FÜR ANGEWANDTE WISSENSCHAFTEN HAMBURG
Hamburg University of Applied Sciences

Nina Cathreen Godenrath

Fallstudienanalyse:

Energieeffizienzsteigerung durch Refactoring

von Java-Anwendungen

Bachelorarbeit eingereicht im Rahmen der Bachelorprüfung
im Studiengang *Bachelor of Science Angewandte Informatik*
am Department Informatik
der Fakultät Technik und Informatik
der Hochschule für Angewandte Wissenschaften Hamburg

Betreuende Prüferin: Prof. Dr. Julia Padberg
Zweitgutachterin: Prof. Dr. Ulrike Steffens

Eingereicht am: 30. November 2023

Nina Cathreen Godenrath

Thema der Arbeit

Fallstudienanalyse: Energieeffizienzsteigerung durch Refactoring von Java-Anwendungen

Stichworte

Refactoring, Java, Nachhaltigkeit, Software, Energieverbrauch, Energieeffizienz, Performance

Kurzzusammenfassung

Diese Bachelorarbeit untersucht die Auswirkungen von Refactorings zur Steigerung der Energieeffizienz von Java-Anwendungen. Sie nutzt durch eine umfangreiche Literaturrecherche ermittelte Refactoring-Techniken und führt Fallstudien mit den Anwendungen GraphStream und LanguageTool durch. Dabei wird nicht nur der Energieverbrauch, sondern auch die Performance berücksichtigt. Die Methodik beinhaltet die Verwendung von Power Gadget zur Energieverbrauchsmessung und die Anwendung von Unit-Tests. Die Ergebnisse werden statistisch analysiert, um die Wirksamkeit der Refactorings und die Beziehung zwischen Performance und Effizienz zu bewerten. Die Arbeit diskutiert auch die Praxisrelevanz ihrer Ergebnisse und weist auf das Potenzial für zukünftige Forschung hin.

Nina Cathreen Godenrath

Title of Thesis

Case Study Analysis: Increasing Energy Efficiency by Refactoring Java Applications

Keywords

Refactoring, Java, sustainability, software, energy consumption, energy efficiency, performance

Abstract

This bachelor's thesis examines the effects of refactorings to increase the energy efficiency of Java applications. It uses refactoring techniques identified through extensive literature research

and conducts case studies with the applications GraphStream and LanguageTool. Not only energy consumption, but also performance is considered. The methodology includes the use of Power Gadget to measure energy consumption and the application of unit tests. The results are statistically analyzed to evaluate the effectiveness of the refactorings and the relationship between performance and efficiency. The work also discusses the practical relevance of its results and points to the potential for future research.

Inhaltsverzeichnis

Abbildungsverzeichnis

Tabellenverzeichnis

Listings

1 Einleitung

1.1 Motivation

In Zeiten zunehmenden Umweltbewusstseins und steigender Energiekosten gewinnt auch die Energieeffizienz von Softwareanwendungen immer mehr an Bedeutung. Dies führt dazu, dass diejenigen, die diese Anwendungen entwickeln, nach Wegen suchen, zu Energieeinsparungen beizutragen. Obwohl häufig besonderes Augenmerk auf der Hardware liegt, besteht auch bereits in der Programmierung ein erhebliches Potenzial, den Energieverbrauch zu optimieren und somit einen positiven Beitrag zur Nachhaltigkeit zu leisten; zumal Softwareentwickler:innen meist keinen oder wenig Einfluss darauf haben, auf welcher Hardware der Code ausgeführt wird. Ein gezieltes Refactoring kann dabei helfen, ineffiziente Softwarekomponenten zu identifizieren und zu optimieren, um den Energieverbrauch zu reduzieren.

Da Java nach wie vor eine der am weitesten verbreiteten Programmiersprachen in der Softwareentwicklung ist, bietet sich diese Sprache besonders für die Untersuchung des Energiesparpotenzials durch Refactorings an. Wenn Refactoring-Maßnahmen identifiziert werden können, die die Energieeffizienz von Java-Anwendungen signifikant verbessern, haben diese Maßnahmen das Potenzial, einen größeren Einfluss auf die Gesamtbilanz des Energieverbrauchs zu nehmen als bei einem Fokus auf unbekannte und kaum genutzte Sprachen.

Eine hohe Performance (im Sinne einer möglichst geringen Laufzeit) hat bereits heute in den meisten Software-Projekten eine hohe Priorität, da dies den Erwartungen der Anwender:innen entspricht. Es erscheint auf den ersten Blick naheliegend, dass Performance und Energieeffizienz miteinander in Zusammenhang stehen. In dem Falle könnte es ausreichend sein, den

Schwerpunkt auf der Performance zu belassen, um Energie einzusparen. Dennoch bedarf es einer gründlichen Prüfung, um festzustellen, ob dies tatsächlich zutrifft.

1.2 Zielsetzung

In Anbetracht der gegenwärtigen Relevanz wissenschaftlicher Forschung im Kontext der energieeffizienten Programmierung verfolgt diese Bachelorarbeit das Ziel zu überprüfen, inwieweit mithilfe wissenschaftlich erprobter Refactoring-Techniken die Energieeffizienz von Java-Anwendungen gesteigert werden kann.

Die meisten bisherigen Forschungsarbeiten in diesem Bereich konzentrieren sich darauf, den Energiebedarf bei Ausführung isolierter Code-Fragmente zu messen. Die Auswirkungen, die ein Refactoring tatsächlich auf eine umfangreiche Anwendung haben kann, können durch diese isolierten Experimente wahrscheinlich nicht realistisch abgebildet werden. Ein signifikanter Unterschied dieser Bachelorarbeit zu diesen Untersuchungen besteht darin, dass mit Hilfe von Unit-Tests versucht wird, der Simulation eines realen Anwendungsszenarios nahezukommen.

Hierzu werden ausgewählte Fallstudien betrachtet, um praxisnahe Erkenntnisse zu gewinnen und konkrete Auswirkungen der Code-Optimierungen auszuwerten. Im Mittelpunkt steht die Fragestellung, wie effektiv das Refactoring bei der Steigerung der Energieeffizienz in Java-Anwendungen tatsächlich ist und ob ein Zusammenhang zwischen Performance und Effizienz besteht.

1.3 Aufbau der Arbeit

Die gesamte Arbeit ist in 6 Kapitel gegliedert. Im nachfolgenden Kapitel wird der Stand der Forschung in den Bereichen nachhaltige Software, der Gegenüberstellung von Performance und Energieeffizienz, Refactoring-Techniken und Energieverbrauchmessung dargelegt, wobei die Refactoring-Techniken mit besonderem Fokus auf Energieverbrauch beleuchtet werden, sodass hier frühzeitig eine Sammlung vielversprechender Maßnahmen erstellt werden kann.

Im dritten Kapitel werden die zwei Java-Anwendungen GraphStream und LanguageTool als Objekte der Fallstudien ausführlich vorgestellt. Dabei werden sie anhand der zuvor definierten Auswahlkriterien auf ihre Eignung hin geprüft und sowohl fachlich als auch technisch ausreichend detailliert beschrieben.

Anschließend wird in Kapitel 4 die Methodik vorgestellt: Mit Hilfe des Energiemesswerkzeugs Power Gadget und Unit-Tests wird ein fester Aufbau für den Messvorgang konstruiert, der zuverlässig wiederholbar vor und nach dem jeweiligen Refactoring durchgeführt werden kann. Das Refactoring wird in den wesentlichen Aspekten dokumentiert. Die Messung des Energieverbrauchs und der Laufzeit erfolgt mehrere Male vor und nach dem Refactoring, um die Auswirkungen der durchgeführten Änderungen im Folgenden zu analysieren.

Die Ergebnisse werden im fünften Kapitel vorgestellt und im Rahmen einer statistischen Analyse hinsichtlich der Wirksamkeit der durchgeführten Refactorings und der Korrelation zwischen Performance und Effizienz diskutiert. Zudem erfolgt eine kritische Bewertung der angewandten Methodik und der einschränkenden Faktoren, denen die Arbeit unterlag.

Abschließend fasst der letzte Abschnitt die wichtigsten Erkenntnisse zusammen und bietet einen Ausblick auf mögliche zukünftige Forschungsrichtungen.

2 Stand der Forschung

Schlagworte wie ‚Digitale Nachhaltigkeit' und ‚Sustainable IT' sind im Kontext der Informationstechnologie in den letzten Jahren immer häufiger anzutreffen. Diese Begriffe lassen sich inhaltlich mit der Bezeichnung ‚Green IT' und ihrer Definition zusammenfassen: Laut Murugesan (2008) beschreibt der Begriff „die Erforschung und Praxis des umweltbewussten Designs, der Herstellung, Nutzung und Entsorgung von Computern, Servern und zugehörigen Subsystemen wie Monitoren, Druckern, Speichergeräten sowie Netzwerk- und Kommunikationssystemen. Dabei wird angestrebt, diese Ressourcen effizient und effektiv zu nutzen, um eine minimale oder gar keine Auswirkungen auf die Umwelt zu erzielen. Green IT zielt auch darauf ab, wirtschaftlich rentabel zu sein und eine verbesserte Systemleistung und Nutzung zu ermöglichen, während gleichzeitig soziale und ethische Verantwortung wahrgenommen wird."

Auch wenn der Fokus auf Green IT in den vergangenen Jahren gewachsen ist und weiterhin steigt, beschränken sich Betrachtungen und Optimierungsansätze häufig auf die Hardware, die zur Ausführung der Software verwendet wird. Tatsächlich gibt es hier großes Optimierungspotenzial in Bezug auf Nachhaltigkeit in allen Stufen ihres Lebenszyklus, von der Produktion bis zur Entsorgung (Gupta et al., 2021).

2.1 Nachhaltige Softwareentwicklung

Das vorhandene Einsparpotenzial bei der Hardware ist kein Grund, nicht auch bereits in der Entwicklung von Software ein besonderes Augenmerk auf Energieeffizienz zu legen. Agarwal et al. (2012) argumentieren, dass sowohl Anwender:innen als auch Entwickler:innen schnell dem Fehlschluss erliegen, Software sei von Hause aus ‚grün', was dazu führe, dass Nachhaltigkeit im gesamten Softwareentwicklungsprozess zu kurz komme, wenn nicht sogar gegen sie

gearbeitet werde. So werde beispielsweise ständig neue, noch leistungsstärkere Hardware eingesetzt, um die Defizite veralteter Software auszugleichen. Auf diese Weise werde langsame, ineffiziente und teure Software lange am Leben erhalten, ohne dass sich nach außen hin Einbußen in der Performance bemerkbar machen.

Agarwal et al. (2012) schlagen einige nachhaltige bewährte Praktiken für die Entwicklung und Implementierung von Software-Systemen vor, darunter auch eher simple und naheliegende Aspekte wie das Verwenden von digitalen statt analogen Fragebögen bei Nutzertests, die Aufnahme von festen Nachhaltigkeitskriterien, die es einzuhalten und möglichst auch durch Tests abzusichern gilt, das Minimieren von Redundanzen und ein Fokus auf energieeffizienten Code.

Auch Georgiou et al. (2020) heben im Rahmen einer umfassenden Analyse mit dem Titel *Software Development Lifecycle for Energy Efficiency: Techniques and Tools* unter anderem die Relevanz von energieeffizientem Code hervor. Die Autor:innen präsentieren hier auf umfassende Weise für jeden einzelnen Schritt des Lebenszyklus von Softwareentwicklung (Requirements, Design, Implementation, Verification, Maintenance) Maßnahmen zur Förderung der Nachhaltigkeit. An dieser Stelle soll nur auf ihre Ergebnisse zu Design und Implementation eingegangen werden, wenngleich die Maßnahmen in den anderen Bereichen nicht minder wichtig sind.

Im Bereich Design wird in der Analyse mehrerer Studien durch Georgiou et al. (2020) deutlich, dass es Design-Patterns (Entwurfsmuster) gibt, die deutlich weniger energieeffizient sind als andere. Als besonders ungünstig heben sie die drei Patterns Observer, Decorator und Abstract Factory hervor. Auf die Energieeffizienz einzelner Design-Patterns wird in Abschnitt 2.3 genauer eingegangen.

Der Bereich Implementation umfasst in der Analyse verschiedene Unterpunkte, darunter die Wahl der Programmiersprache(n), Datenstrukturen und Programmiertechniken.

Von den fünf miteinander verglichenen Programmiersprachen erweisen sich Assembler, C und C++ als wesentlich effizienter als Python und Java. Diese Bewertung deckt sich mit den Erkenntnissen von Pereira et al. (2021), wenngleich nach den Messungen von Pereira et al. Python schlechter als Java abschneidet, während es in den von Georgiou et al. analysierten Studien andersherum ist. Deutlich wird in jedem Fall, dass die Wahl der Programmiersprache ein

wichtiger Faktor für energieeffiziente Software ist. Trotz dieser Erkenntnisse liegen Java-Anwendungen im Fokus dieser Arbeit, da sich in der Praxis ein Refactoring wesentlich leichter umsetzen lässt als der Wechsel auf eine andere, möglicherweise effizientere Programmiersprache.

Auch ein Vergleich der Datenstrukturen lässt relevante Unterschiede in der Energieeffizienz erkennen. Unter anderem wird hier die Arbeit von Hasan et al. (2016) herangezogen, deren Ergebnisse zeigen, dass die Wahl der energieeffizientesten Datenstruktur vom jeweiligen Anwendungsfall abhängt. Zum Beispiel hat sich gezeigt, dass bei der Implementierung von Listen im Java Collection Framework (JCF) die Verwendung von LinkedList beim Einfügen am Anfang energieeffizienter ist, während beim Einfügen in der Mitte oder am Ende ArrayList bevorzugt wird. Die Ergebnisse von Hasan et al. (2016) werden im Abschnitt über Refactoring-Techniken noch einmal aufgegriffen.

Zuletzt seien im Bereich Implementation aus der Analyse von Georgiou et al. (2020) noch die Programmiertechniken genannt. Hier hat sich beispielsweise gezeigt, dass das Ersetzen von Gettern und Settern durch direkten Zugriff auf Klassenvariablen Energie einsparen kann. Diese und weitere Maßnahmen zur Steigerung der Energieeffizienz werden im Abschnitt 2.3 dieser Arbeit gesammelt und erläutert. Zunächst soll aber noch kurz auf den Zusammenhang zwischen Performance und Energieeffizienz eingegangen werden.

2.2 Performance vs. Energieeffizienz

Als eines der Qualitätsattribute, die direkt von Anwender:innen wahrgenommen werden und somit zu Kauf- oder Nutzungsentscheidungen beitragen können, nimmt die Performance einen hohen Stellenwert in der Softwareentwicklung ein. ‚Performance' bedeutet im Kontext dieser Arbeit, dass die Anwendungen möglichst kurze Antwort-, Verarbeitungs- und Ladezeiten aufweisen.

Der Begriff der Effizienz ist etwas vielseitiger und kann sich auf unterschiedliche Aspekte beziehen: Ein performantes System kann zunächst als effizient angesehen werden, insbesondere im Hinblick auf die Zeit. Wenn ein Programm einen geringen Energieverbrauch aufweist, wird

es als energieeffizient bezeichnet. Ebenso kann die Effizienz in Bezug auf den Speicherplatz betrachtet werden. Der Fokus in dieser Bachelorarbeit wird auf der Energieeffizienz und ihrer Relation zur Performance (Zeiteffizienz) liegen. Effizienz in Bezug auf Speicherplatz ist nicht Gegenstand dieser Arbeit.

Energieeffizienz wird häufig mit CO_2-Einsparungen assoziiert. Da zur Berechnung des CO_2-Fußabdrucks von Software aber sinnvollerweise weitere Faktoren wie die Hardware und die Stromquelle berücksichtigt werden sollten, wird das Ausmaß an potenziellen CO_2-Einsparungen in dieser Arbeit nicht weiter betrachtet.

Zudem muss zwischen Leistung und Energie unterschieden werden: Die Leistung eines elektrischen Geräts wird in Watt (W) angegeben. Diese Einheit zeigt an, wie viel Energie in einer bestimmten Zeitspanne verbraucht wird. Elektrische Energie wird meist in Kilowattstunden (kWh) oder in Joule (J) gemessen und gibt die gesamte Menge an Strom an, die über einen bestimmten Zeitraum verbraucht wird. Die Ergebnisse der Energiemessungen im experimentellen Teil dieser Arbeit werden in Joule angegeben.

Es gilt also $Energie = Zeit * Leistung$.

Ein naheliegender Denkansatz ist an dieser Stelle, dass Software, die zeiteffizient ist, auch energieeffizient sein muss. In einer Studie, die sich mit dem Zusammenhang zwischen Zeit, Energie und Speicher bei unterschiedlichen Programmiersprachen beschäftigt, kommen Pereira et al. (2017) allerdings zu dem Schluss, dass die Programmiersprache mit der geringsten Laufzeit nicht immer am wenigsten Energie verbraucht. Sie argumentieren in Bezug auf ihre Ergebnisse, dass der Faktor Leistung aus der oben genannten Formel nicht zwingend konstant ist. Gleichzeitig weisen sie darauf hin, dass es sowohl Untersuchungen gibt, die für einen Zusammenhang zwischen Energieeffizienz und Performance sprechen (Yuki & Rajopadhye, 2014) als auch solche, die zum gegenteiligen Schluss kommen (Trefethen & Thiyagalingam, 2013; Pinto et al., 2014; Lima et al., 2016).

Tatsächlich beziehen sich die Untersuchungen zu dieser Thematik meist auf den Vergleich unterschiedlicher Compiler, selten auf Programmiersprachen und noch seltener auf konkrete Programmiertechniken. So beschränkt sich auch die Überzahl der im folgenden Abschnitt behandelten Studien zu nachhaltigen Refactoring-Techniken häufig auf die Messung des

Energieverbrauchs in Joule (ohne Zeitmessung). Es bleibt also zu untersuchen, ob sich anhand konkreter Beispiele eine eindeutige Korrelation zwischen Performance und Energieeffizienz beim gezielten Refactoring von Java-Anwendungen erkennen lässt. Aus diesem Grund wird in dieser Arbeit neben dem Energieverbrauch auch die Laufzeit als Messgröße betrachtet.

Von den wenigen Studien, in denen die Zeit ebenfalls gemessen und angegeben wurde, lässt sich hier jene von Pereira et al. (2016) hervorheben: Sie haben Optimierungen durch den Austausch von Collection-Datenstrukturen vorgenommen (hauptsächlich wurden hier LinkedList durch ArrayList und HashMap durch HashTable ersetzt) und Energieeinsparungen erzielt (siehe Tabelle 1).

Projects	**Data Structures**					
	Original		**Optimized**		**Improvement**	
	J	ms	J	ms	J	ms
1	23.744583	1549	22.7071302	1523	4.37%	1.68%
2	24.6787895	1823	23.525123	1741	4.67%	4.50%
3	25.0243507	1720	22.259355	1508	11.05%	12.33%
4	17.1994425	1258	16.2014997	1217	5.80%	3.26%
5	19.314512	1372	18.3067573	1245	5.22%	9.26%

Tabelle 1: Optimierungsergebnisse von Pereira et al. (2016).

In der Spalte Improvement ist zu sehen, dass in dieser Studie Energie- und Zeitverbesserungen nicht immer korrelieren. Während in den Projekten 2 und 3 Energie- und Zeiteffizienz etwa im gleichen Maße verbessert wurden, ist bei Projekt 1 der Effekt auf den Energieverbrauch höher und bei Projekt 5 ist es andersherum. Die Autor:innen schließen hieraus, dass anhand ihrer Ergebnisse keine Relation zwischen Energieverbrauch und Laufzeit festgestellt werden kann.

Bezüglich des Begriffs ‚Energieverbrauch' soll hier nicht unerwähnt bleiben, dass Energie streng genommen nicht verbraucht, sondern vielmehr umgewandelt wird. Um jedoch im Vokabular der Referenzliteratur zu bleiben, in der sehr häufig der Begriff ‚energy consumption' anzutreffen ist, wird das Wort ‚Energieverbrauch' verwendet, um die Menge an Energie zu beschreiben, die zur Ausführung einer Anwendung benötigt wird.

2.3 Refactoring-Techniken

In seinem erstmals 1999 erschienen Werk *Refactoring: Improving the Design of Existing Code*, das noch heute als Standardwerk in Bezug auf Softwareentwicklung und Refactoring gilt, gibt Martin Fowler für den Begriff ‚Refactoring' eine Definition an, die noch heute gültig ist und häufig herangezogen wird: „Refactoring ist der Prozess, ein Softwaresystem so zu modifizieren, dass sich das externe Verhalten des Codes nicht ändert, aber dennoch die interne Struktur des Codes zu verbessern." (Fowler, 2020, S. 20)

Im Kontext seines Buches bedeutet ‚verbessern' für Fowler primär, dass der Code verständlicher wird. Er grenzt den Begriff klar gegen Performance-Optimierungen ab. Der Unterschied liege in dem Ziel: Obwohl bei beiden Verfahren der Code modifiziert werde, ohne dessen Funktionalität zu verändern, könne bei einem Refactoring das Programm dabei schneller oder langsamer werden, während bei Performance-Optimierungen das Programm beschleunigt werden solle, selbst wenn der Code dadurch unhandlicher werden könne. (Fowler, 2020, S. 80)

Trotz dieser Abgrenzung wird der Begriff ‚Refactoring' heutzutage in der Softwareentwicklung verwendet, um jegliche Code-Änderungen zu beschreiben, die die Funktionalität nicht verändern, den Code aber in irgendeiner Weise verbessern sollen; wobei diese Verbesserung unterschiedliche Ziele hat. Demnach würde man auch bei Performance-Optimierungen von einem ‚Refactoring für Performance' sprechen und analog bei Anpassungen des Programmiercodes zur Reduzierung des Energieverbrauchs von ‚Refactoring für Energieeffizienz'. Dies zeigen auch die Titel zahlreicher in dieser Arbeit herangezogener Veröffentlichungen wie beispielsweise *Code refactoring techniques for reducing energy consumption in embedded computing environment* von Kim et al. (2018), *Energy Efficiency Analysis of Code Refactoring Techniques for Green and Sustainable Software in Portable Devices* von Şanlıalp et al. (2022) und *Improving Energy Efficiency Through Automatic Refactoring* von Cruz & Abreu (2019).

Durch Refactorings soll also stets der Code hinsichtlich eines bestimmten Aspekts verbessert werden, ohne das äußerliche Verhalten zu ändern. Fowlers Abgrenzung macht deutlich, dass der Fokus auf einen Aspekt zur Folge haben kann, dass ein anderer darunter leidet. So wird auch in der vorliegenden Arbeit an einigen Stellen der Code zugunsten der Energieeffizienz verändert, obwohl dies zu Einbußen in der Übersichtlichkeit führt.

In der Analyse von Refactoring-Techniken im Kontext der Energieeffizienz liegt der Fokus der meisten Studien und Forschungsarbeiten vorrangig auf mobilen Geräten. Dieser Fokus ergibt sich aus der zunehmenden Bedeutung der Energieeffizienz in mobilen Anwendungen, da mobile Geräte oft über begrenzte Akkukapazitäten verfügen und eine optimale Nutzung der Energie erforderlich ist, um eine längere Akkulaufzeit zu gewährleisten. Dieser Umstand kommt der vorliegenden Arbeit zugute, da viele mobile Anwendungen in der Programmiersprache Java geschrieben sind.

Im Folgenden werden sowohl Refactoring-Techniken untersucht, die speziell darauf abzielen, die Energieeffizienz von Java-Anwendungen zu verbessern, als auch solche, die auf andere Programmiersprachen anwendbar sind (wie beispielsweise das Vereinfachen verschachtelter Schleifen oder Entfernen von Gettern und Settern). Diese Techniken werden gesammelt und bei Verwendung in Kapitel 4 noch einmal zusammengefasst aufgelistet, um beim Refactoring im praktischen Teil dieser Bachelorarbeit eingesetzt zu werden.

Die Refactoring-Techniken sind der Übersicht halber in unterschiedlichen Kategorien zusammengefasst, die sich jeweils auf einen Aspekt des Codes beziehen: Design, Kontrollstrukturen, Datenstrukturen und (Weitere) Energy Smells.

2.3.1 Design

Die Analyse von Georgiou et al. (2020) listet einige vielversprechende Refactoring-Techniken insbesondere bezüglich Design auf. Sie zeigen, dass die Optimierung von ineffizienten Design-Patterns wie Observer und Decorator den Energieverbrauch im Schnitt um 10% senken kann. In einer aktuellen Arbeit konnten Bree & Cinneide (2022) ebenfalls durch Code-Optimierungen im Kontext des Decorator-Patterns den Energieverbrauch der Open-Source-Anwendung JUnit um 5% senken. Unter ‚Optimierung' wird hier vor allem die Reduzierung von Objekten und Funktionsaufrufen durch Auflösung dieser Patterns verstanden. Ob sich durch die Anpassung oder gar das Entfernen von Design-Patterns die Energieeffizienz verbessern lässt, hängt natürlich davon ab, welche Patterns überhaupt verwendet werden.

Hieraus ergeben sich folgende Techniken, die bezüglich Design-Patterns angewendet werden sollten (unter der Voraussetzung, dass eines der potenziell problematischen Design-Pattern Observer, Decorator oder Abstract Factory verwendet werden):

- Reduzierung von Objekten
- Reduzierung von Funktionsaufrufen

Design-Patterns stellen wiederverwendbare Lösungen für bestimmte Probleme dar und sind nicht auf eine Programmiersprache beschränkt. Im Gegensatz dazu beziehen sich Konzepte wie Vererbung, Kapselung und Polymorphie auf grundlegende Prinzipien der objektorientierten Programmierung. Diese Konzepte sind integraler Bestandteil einer Programmiersprache und dienen dazu, die Struktur und das Verhalten von Objekten zu organisieren und zu verwalten.

Auch in der Wahl und Ausgestaltung dieser Konzepte lässt sich Energiesparpotenzial erkennen: Bree & Cinnéide (2020) ersetzten in eigens für ihre Experimente geschriebenen Code-Snippets Delegation durch Vererbung, was zu einer Verringerung des Energieverbrauchs um 81% führte.

Der Unterschied zwischen Delegation und Vererbung wird in Listing 1 und 2 deutlich. In Listing 1 werden bei Objekten der Klasse `Employee` alle Aufrufe, welche die Klasse `Person` betreffen, an ein `Person`-Objekt delegiert. Ein Aufruf von `getName()` auf einem `Employee`-Objekt würde also wiederum `getName()` auf einem `Person`-Objekt aufrufen.

```
public class Person {
  public Person(String name) {
    this.name = name;
  }

  public String getName() {
    return name;
  }

 private String name;
}
```

```
public class Employee {
  public Employee(String name) {
    person = new Person(name);
  }

  public String getName() {
    return person.getName();
  }

 private Person person;
}
```

Listing 1: Java-Implementierung einer Delegation von Bree & Cinnéide (2020).

In Listing 2 erbt die Klasse `Employee` von der Klasse `Person`. Objekte der Klasse `Employee` verfügen also über die Methoden und Variablen von `Person` (und in der Regel außerdem noch über weitere eigene). Ein Aufruf von `getName()` auf einem `Employee`-Objekt wird also nicht delegiert, sondern gibt den Wert der Variable `name` direkt zurück, da die `getName`-Methode der Klasse `Person` aufgerufen wird.

```
public class Person {
  public Person(String name) {
    this.name = name;
  }

  public String getName() {
    return name;
  }

 private String name;
}

public class Employee extends Person {
  public Employee(String name) {
```

```
        super(name);
    }
}
```

Listing 2: Java-Implementierung einer Vererbung von Bree & Cinnéide (2020).

Anhand der Listings 1 und 2 wird bereits deutlich, worin auch die Autor:innen den Grund für den starken Unterschied im Energieverbrauch von Delegation und Vererbung sehen: Durch Delegation kann sich die Anzahl der Methodenraufrufe verdoppeln.

An dieser Stelle wird erkennbar, wie sich Refactoring für mehr Verständlichkeit und Refactoring für Energieeffizienz widersprechen können: Fowler empfiehlt in seinem Buch, bei komplexer werdenden Vererbungshierarchien zu Delegation zu wechseln, da Vererbung für eine zu enge Kopplung sorge und zu wenig Raum für Variationen biete (Fowler, 2020, S. 431).

Ähnlich verhält es sich bei dem Konzept der Kapselung. Im Sinne der Entkopplung und besseren Lesbarkeit von Code rät Fowler zur Kapselung, also beispielsweise der Verwendung von Gettern und Settern bei Variablen. Dies mache Änderungen leichter und zudem sei es für den Benutzer eines Objekts nicht nötig zu wissen, ob ein Wert berechnet oder gespeichert werde (Fowler, 2020, S. 204). Tonini et al. (2013) zeigen jedoch, dass durch das Entfernen von Gettern und Settern in einer in Java programmierten Smartphone-Anwendung sowohl Energie- als auch Zeitaufwand verringert werden können.

Als weitere Refactoring-Techniken lassen sich also festhalten:

- Ersetzen von Delegation durch Vererbung
- Entfernen von Gettern und Settern

In derselben Arbeit untersuchen Tonini et al. (2013) auch die Energiesparpotenziale bei der Verwendung unterschiedlicher Implementierungen von for-Schleifen. Um diese und um andere Kontrollstrukturen geht es im folgenden Abschnitt.

2.3.2 Kontrollstrukturen

Tonini et al. (2013) vergleichen drei Arten, eine for-Schleife für die Iteration über Collections zu implementieren: Erstens ein for-each-Konstrukt, zweitens eine for-Schleife, die die Länge zuvor speichert, und drittens eine for-Schleife, die die Länge in jeder Iteration abfragt. Beispiele für diese Implementierungen sind in Listing 3 zu sehen. Die Versuche wurden mit der Collection-Datenstruktur ArrayList durchgeführt.

```
ArrayList<Employee> employees = getEmployees();

// for-each-Konstrukt
for (Employee employee : employees) {
  System.out.print(employee.getName());
}

// Schleife, die die Länge speichert
int size = employees.size();
for (int i = 0; i < size; i++) {
  System.out.print(employees.get(i).getName());
}

// Schleife, die die Länge in jeder Iteration abfragt
for (int i = 0; i < employees.size(); i++) {
  System.out.print(employees.get(i).getName());
}
```

Listing 3: Drei mögliche Implementierungen einer `for`-Schleife in Java.

Für die Iteration über Collections vom Typ ArrayList ist laut den Ergebnissen der Autor:innen die Implementierung am effizientesten, die zuerst die Länge speichert und dann in einer klassischen indizierten for-Schleife die Elemente durchgeht. Die Wahl der Iterationsimplementierung hängt allerdings von der jeweiligen Datenstruktur ab. Tonini et al. (2013) weisen selbst darauf hin, dass das for-each-Konstrukt zwar bei ArrayList am meisten Energie verbrauche, dies jedoch nicht für alle Collections gelte. Es handele sich bei ArrayList vielmehr um einen Sonderfall. Dies wird in Abschnitt 2.3.3 noch einmal aufgegriffen.

Für alle anderen Arten von Collections ist daher wahrscheinlich das for-each-Konstrukt vorzuziehen: Palomba et al. (2019) identifizieren in einer Studie, die sich auf Android-Anwendungen fokussiert, vier Energy Smells (in Anlehnung an den Begriff Code Smells; mehr dazu unter 2.3.4), die starke Auswirkungen auf die Energieeffizienz haben, darunter auch klassische for-Schleifen. Sie verbessern den Energieverbrauch, indem sie diese durch for-each-Konstrukte ersetzen. Der zweite der vier Energy Smells sind Setter (siehe oben unter 2.3.1). Die beiden verbleibenden werden weiter unten in Abschnitt 2.3.4 besprochen.

Ein weiterer wichtiger Aspekt in Bezug auf den Energieverbrauch bei for-Schleifen ist die Verschachtelung: Sowohl Kim et al. (2018) als auch Şanlıalp et al. (2022) konnten zeigen, dass mit der Vereinfachung verschachtelter Schleifen der Energieverbrauch eindeutig gesenkt werden kann. Listing 4 zeigt ein Beispiel einer solchen Vereinfachung nach Şanlıalp et al. (2022).

```
public int nestedLoop(int row, int column, int iterations) {
  int result = 0;
  for (int i = 0; i < row; i++) {
    for (int j = 0; j < column; j++) {
      for (int k = 0; k < iterations; k++) {
        result += row * column;
      }
    }
  }
  return result;
}

public int simplifiedLoop(int row, int column, int iterations) {
  int result = 0;
  for (int i = 0; i < row * column * iterations; i++) {
    result += row * column;
  }
    return result;
}
```

Listing 4: Beispielhafte Vereinfachung einer verschachtelten Schleife nach Şanlıalp et al. (2022).

Die beiden Schleifen-Implementierungen in Listing 4 errechnen bei gleichen Parameterwerten denselben Wert. Allerdings verbraucht den Ergebnissen von Kim et al. (2018) und Şanlıalp et al. (2022) zufolge die Methode `nestedLoop` mehr Energie als die Methode `simplifiedLoop`.

Bei der `nestedLoop`-Methode wird die äußere Schleife wird `row` Mal durchlaufen, die innere Schleife wird `column` Mal pro äußerer Schleifendurchlauf durchlaufen und die innerste Schleife wird `iterations` Mal pro innerem Schleifendurchlauf durchlaufen. Das bedeutet, dass insgesamt `row * column * iterations` Schleifendurchläufe stattfinden und genauso häufig die Zeile `result += row * column;` ausgeführt wird.

Im Gegensatz dazu hat die `simplifiedLoop`-Methode nur eine einzige Schleife, die ebenfalls `row * column * iterations` Mal durchlaufen wird. Auch hier wird in jedem Schleifendurchlauf `result += row * column;` ausgeführt.

Obwohl beide Methoden dieselbe Anzahl von Schleifendurchläufen und Additionen ausführen, verbraucht die `nestedLoop`-Methode mehr Energie und Rechenleistung, vermutlich weil sie zusätzlich die Verwaltung und Ausführung der temporären Variablen `j` und `k` und verschachtelten Schleifen erfordert. Es erscheint daher sinnvoll, verschachtelte Schleifen möglichst zu vereinfachen.

Bei dem in Listing 4 dargestellten Code, der auf Şanlıalp et al. (2022) basiert, handelt es sich natürlich um ein sehr konstruiertes Beispiel, das nur auf diese Weise möglich ist, wenn die Berechnungen in der innersten Schleife Konstanten (beziehungsweise von den Indizes `i`, `j` und `k` unabhängig) sind. Es ist sogar diskutabel, ob hier der Begriff der Verschachtelung überhaupt angemessen ist, da lediglich überflüssige Schleifen verwendet werden. Die Untersuchungen von Kim et al. (2018) fokussieren sich auf die Programmiersprache C. Für diese Sprache schlagen sie zur Vereinfachung verschachtelter Schleifenstrukturen vor, sie durch eine Struktur zu ersetzen, die Array-Zeiger und eine einzelne Schleife verwendet, sofern dies möglich ist. Das ist in Java allerdings nicht möglich und daher im Kontext dieser Arbeit keine Option. Die Handlungsanweisung, überflüssige Schleifen aufzulösen, bleibt jedoch auch für Java und trotz des zweifelhaften Beispiels von Şanlıalp et al. (2022) bestehen, da die Ergebnisse beider Arbeiten deutlich zeigen, dass sie der Energieeffizienz zuträglich ist.

Bezüglich for-Schleifen lassen sich aus den oben erläuterten Ergebnissen folgende Refactoring-Techniken ableiten:

- Verwendung von for-each-Konstrukt bei allen Collections außer ArrayList
- Verwendung von for-Schleife mit Länge in externer Variable bei ArrayList
- Vereinfachung verschachtelter Schleifen

Bezüglich der Energieeffizienz bei Bedingungen und Operatoren kamen Kumar et al. (2017) aufgrund ihrer Messungen zu folgendem Ergebnis: Wenn eine Bedingung aus mehreren Teilen besteht, wird mehr Energie gespart, je früher einer dieser Teile die gesamte Bedingung entscheidet. Listing 5 zeigt den von Kumar et al. (2017) verwendeten Code.

```
int a = 0;
for (int i = 0; i < 2000000000; i++) {
  if (a > -1 || a > 1 || a > 1) {
    a += 1;
  }
}

int a = 0;
for (int i = 0; i < 2000000000; i++) {
  if (a < -1 || a < -1 || a > -1) {
    a += 1;
  }
}
```

Listing 5: Code-Snippet zur Energieverbrauchsmessung bei Operatoren in Java nach Kumar et al. (2017).

Da beide Schleifen in Listing 5 mit `a = 0` starten und `a` anschließend nur noch inkrementiert wird, wird in der oberen Schleife stets nur der erste Teil der Bedingung ausgewertet. Das liegt daran, dass diese mit dem Rest der Bedingung durch einen Operator verknüpft ist, der `true` zurückgibt, sobald mindestens eine Bedingung erfüllt ist; und `a` ist in dieser Implementierung immer größer als -1.

Die untere Schleife wertet bei jedem Durchgang alle drei Teile der Bedingung aus, da die ersten beiden stets mit `false` und erst die dritte mit `true` ausgewertet wird.

Nach den Messungen von Kumar et al. (2017) benötigte die untere Schleife 10% mehr Energie als die obere. Eine analoge Implementierung mit dem &&-Operator erbrachte dasselbe Ergebnis im Sinne seiner Funktionsweise. Eine intelligente Wahl der Reihenfolge innerhalb von Bedingungen ist also anzuraten und kann als Refactoring-Technik festgehalten werden:

- Anpassung der Reihenfolge innerhalb von Bedingungen (sodass der wahrscheinlichste Fall frühzeitig die gesamte Bedingung entscheidet)

2.3.3 Datenstrukturen

In *Energy Profiles of Java Collection Classes* vergleichen Hasan et al. (2016) verschiedene Java-Implementierungen der Datenstrukturen List, Map und Set im Hinblick auf ihre Energieeffizienz und kommen zu dem Schluss, dass die Frage, welche Datenstruktur und Implementierung am wenigsten Energie verbraucht, nicht allgemeingültig beantwortet werden kann und immer vom Kontext abhängt. Sie zeigen außerdem, dass die Wahl einer ungeeigneten Datenstruktur den Energieverbrauch um über 300% steigern kann (wenngleich sie selbst darauf hinweisen, dass dies einen Extremfall darstellt).

Library	**List**	**Map**	**Set**
Java Collections Framework (JCF)	ArrayList LinkedList	HashMap TreeMap	HashSet TreeSet LinkedHashSet
Apache Collections Framework (ACC)	TreeList	HashedMap LinkedMap	ListOrderedSet MapBackedSet
Trove	TIntArrayList TIntLinkedList	TIn- tIntHashMap	TIntHashSet

Tabelle 2: Von Hasan et al. (2016) untersuchte Datenstrukturen.

Die untersuchten Datenstrukturen und die Librarys mit den unterschiedlichen Implementierungen sind in Tabelle 2 abgebildet. Abbildung 1 zeigt die Resultate der Energieverbrauchmessungen dieser Implementierungen für bestimmte Anwendungsfälle graphisch übersichtlich aufbereitet. Eine grüne Einfärbung steht hier für geringen Energieverbrauch, eine rote für einen hohen. Hier wird ersichtlich, was in Abschnitt 2.1 bereits kurz erwähnt wurde: Die LinkedList-Implementierung des JCF verhält sich beim Einfügen am Anfang einer Liste wesentlich energieeffizienter als die ArrayList-Implementierung, während ArrayList-Implementierungen grundsätzlich weniger Energie beim Einfügen in der Mitte und am Ende benötigen. Aus (a) in Abbildung 1 wird auch deutlich, dass die TreeList des ACC bei Fokus auf Energieeinsparungen vermieden werden sollte.

Hasan et al. (2016) verwendeten bei ihren Experimenten für die Iteration eine for-Schleife mit Zählvariable. Wie in Abschnitt 2.3.2 anhand der Ergebnisse von Tonini et al. (2013) ermittelt wurde, ist diese Variante nur für die Datenstruktur ArrayList, nicht aber für die anderen die effizienteste Wahl. Dies spiegelt sich in der Matrix (a) in Abbildung 1 wider, in der die Iteration über ArrayList weniger Energie benötigte als bei den übrigen Datenstrukturen. Es bleibt zu vermuten, dass die Ergebnisse von Hasan et al. (2016) bezüglich des Energieverbrauchs bei Iterationen unter Verwendung eines for-each-Konstruktes anders ausgesehen hätten und die Iteration über ArrayList mehr Energie benötigt hätte.

Matrix (b) bezüglich der Map-Implementierungen zufolge sind fast alle Implementierungen gleich energieeffizient, wobei jedoch die LinkedMap des ACC und die TreeMap des JCF nicht optimal sind und daher bei einem Refactoring gegen eine der anderen Implementierungen eingetauscht werden sollten.

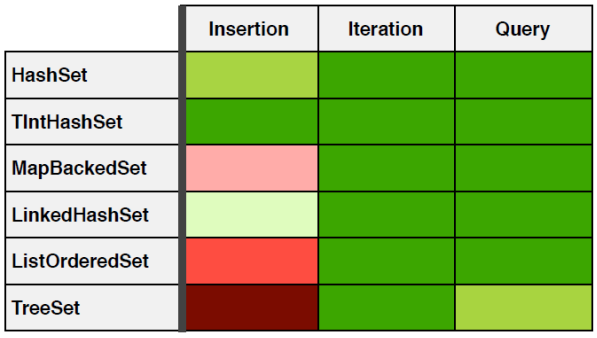
Das Einfügen scheint bei den meisten Set-Implementierungen in Matrix (c) in Abbildung 1 eher energieintensiv zu sein. Hier erscheint es sinnvoll, die Implementierung mit dem Namen TIntIntHashMap der performance-orientierten Library Trove zu verwenden, da diese bei allen untersuchten Operationen auf Maps (Einfügen, Iterieren, Abfragen) in Relation zu den anderen Implementierungen am wenigsten Energie verbraucht.

	Insertion			Iteration	Random Access
	At Beginning	At Middle	At End		
ArrayList					
TIntArrayList					
LinkedList					
TIntLinkedList					
TreeList					

(a) List Matrix

	Insertion	Iteration	Query
HashMap			
TIntIntHashMap			
HashedMap			
LinkedHashMap			
LinkedMap			
TreeMap			

(b) Map Matrix



	Insertion	Iteration	Query
HashSet			
TIntHashSet			
MapBackedSet			
LinkedHashSet			
ListOrderedSet			
TreeSet			

(c) Set Matrix

Abbildung 1: Energieeffizienz der verschiedenen Datenstrukturenimplementierung nach Hasan et al. (2016)

Bezüglich der Library Trove muss erwähnt werden, dass sie zwar Listen-Implementierungen für die Datentypen byte, char, double, float, int, long und short anbietet, nicht aber für Objekte. Das Ersetzen durch eine Listenimplementierung von Trove ist also nur möglich, wenn in der Liste einer dieser Datentypen gespeichert wird. Für Sets und Maps gibt es Implementierungen für die Speicherung von Objekten, die über die primitiven Datentypen hinausgehen.

Die Ergebnisse der Arbeit von Hasan et al. (2016) in Kombination mit denen von Pereira et al. (2016) bieten eine Grundlage für ein Refactoring in Bezug auf die verwendeten Datenstrukturen. Pereira et al. (2016) haben ebenfalls Collection-Datenstrukturen in Java betrachtet und die verwendeten Implementierungen je nach Kontext im Sinne der Energieeffizienz sinnvoll ausgetauscht. Es gibt nur zwei Ergänzungen, die sich aus ihrer Studie zu den aus Abbildung 1 abgeleiteten Handlungsanweisungen ergeben: Erstens ist es sinnvoll, HashMap durch die Datenstruktur-Implementierung Hashtable zu ersetzen, die Hasan et al. (2016) in ihrer Studie nicht betrachtet haben. Zweitens geht aus den Resultaten von Pereira et al. (2016) hervor, dass die nebenläufigen Datenstrukturen ConcurrentSkipListSet, CopyOnWriteArrayList, ConcurrentHashMap, ConcurrentSkipListMap und CopyOnWriteArraySet im Verlgeich zu den nicht-nebenläufigen Implementierungen einen höheren Energieverbrauch aufweisen und demnach bei einem Refactoring mit dem Ziel der Steigerung der Energieeffizienz ausgetauscht werden sollten.

Konkret ergeben sich für ein Refactoring für Energieeffizienz also folgende Handlungsanweisungen:

- Kontextabhängiger Austausch von Collection-Datenstrukturen (Beispiel: Lieber LinkedList statt ArrayList, wenn die Datenstruktur eher wie ein Stack verwendet werden soll und hauptsächlich am Anfang eingefügt wird)
- Austausch von Datenstrukturimplementierungen (Beispiel: TreeList von ACC durch andere Liste einer anderen Library austauschen)
- Ersetzen nebenläufiger Datenstrukturen durch nicht-nebenläufige (sofern der Kontext es erlaubt)

Zuletzt sei zur Arbeit von Hasan et al. (2016) auf etwas hingewiesen, was sie selbst nicht allzu ausführlich thematisieren, für diese Bachelorarbeit jedoch von Relevanz ist: Obwohl nur die verbrauchte Energie dokumentiert wurde, fiel auf, dass ein höherer Energieverbrauch häufig

mit einer längeren Laufzeit einherging, woraus sich schließen ließe, dass hier Performance und Energieeffizienz korrelieren. Leider ist dies aber abseits weniger kurzer Erwähnungen im Text nicht dokumentiert.

2.3.4 (Weitere) Energy Smells

Der Gebrauch des Begriffs Refactoring im Kontext der Energieeffizienzsteigerung wurde weiter oben bereits etabliert. Ein weiterer Begriff, den Fowler und sein Co-Autor Beck prägten, ist ‚Code Smell'. Er beschreibt „Strukturen im Code […], die es nahelegen – oder uns manchmal auch mit der Nase darauf stoßen –, dass ein Refactoring angebracht ist" (Fowler, 2020, S. 107).

Code Smells sind also das, was es durch Refactoring zu beseitigen oder so zu modifizieren gilt, dass der Code dem aktuellen Ziel entspricht. Ähnlich wie der Begriff Refactoring um spezifische Zielvorgaben (Energieeffizienz, Performance) erweitert wurde, ist in der Literatur in diesem Kontext der Begriff ‚Energy Smells' entstanden, um analog zu Code Smells Strukturen zu beschreiben, deren Refactoring zu weniger Energieverbrauch führen kann. Solche Strukturen sind den bisher genannten Kategorien häufig nicht eindeutig zuzuordnen und werden daher hier gesammelt.

Zwei der Energy Smells, die Palomba et al. (2019) identifiziert haben, wurden weiter oben bereits genannt. Die beiden anderen folgen an dieser Stelle.

Der erste betrifft das Starten eines Threads, ohne ihn explizit zu beenden. Als Refactoring-Technik diesbezüglich führen sie eine regelmäßige Überprüfung ein, ob der Thread noch aktiv ist.

Bei dem letzten der vier Energy Smells mit hohem Wirkungsgrad von Palomba et al. (2019) handelt es sich um nicht-statische Methoden, die auf keine Instanzvariablen innerhalb der Klasse zugreifen, in der sie deklariert sind. Statische Methoden zeichnen sich in Java dadurch aus, dass sie aufgerufen werden können, ohne dass eine Instanz der Klasse erstellt werden muss, da sie nicht auf die Eigenschaften einer solchen Instanz zugreifen müssen. Eine Methode, die also statisch sein *könnte*, es aber nicht *ist*, zwingt somit die Aufrufenden dazu, eine Instanz

der ganzen Klasse zu erzeugen, obwohl dies nicht nötig wäre. Als Refactoring-Technik hierfür werden bei Palomba et al. (2019) solche nicht-statischen Methoden in statische umgewandelt.

Von Palomba et al. (2019) werden also zwei weitere Techniken übernommen:

- Überprüfen (und gegebenenfalls Beenden) von Threads, die bisher nicht explizit beendet werden
- Umwandlung nicht-statischer Methoden in statische, wenn sie auf keine Instanzvariablen ihrer Klasse zugreifen

Hassan et al. (2017) zählen ebenfalls einige Refactoring-Techniken zur Beseitigung von Energy Smells auf, die sie aus unterschiedlichen Arbeiten zusammengetragen haben. Viele beziehen sich spezifisch auf die Programmiersprache C++, jedoch sind auch solche darunter, die als allgemeingültig angesehen und somit in diesem Kontext angewendet werden können, nämlich:

- Entfernen von ‚totem' (ungenutztem) Code
- Speichern von Berechnungsergebnissen, die häufiger als einmal verwendet werden, in Variablen (statt mehrfacher Berechnung)

Kumar et al. (2017) haben sich mit dem Energieverbrauch unterschiedlicher Aspekte von Java-Code beschäftigt, indem sie verschiedene Ausprägungen dieser Aspekte (beispielsweise die Verwendung unterschiedlicher primitiver Datentypen für die Repräsentation einer Zahl) in isolierten Code-Snippets implementiert und mit einer hohen Anzahl an Iterationen simple Operationen ausgeführt haben. Die verbrauchte Energie und Laufzeit wurden dabei gemessen und dokumentiert. Ihre Ergebnisse bezüglich der Reihenfolge innerhalb von Bedingungen wurden weiter oben in Abschnitt 2.3.2 bereits präsentiert. Einige weitere sollen hier kurz dargelegt werden, um aus ihnen weitere Refactoring-Techniken abzuleiten.

Von den primitiven Datentypen byte, short, int, long, float, double, char und boolean erwiesen sich int, long und boolean als drei- bis sechsmal energieeffizienter als byte, short, float, double und char. Bei Zahlendarstellungen sind also int und long zu bevorzugen, wenn der Kontext es erlaubt.

Statische Variablen benötigten bei Kumar et al. (2017) 60% mehr Zeit und Energie als nicht-statische. In beiden Fällen wurde die Variable außerhalb einer Schleife deklariert und dann in 2.000.000.000 Iterationen hochgezählt. Die Autor:innen deuten an, dass der höhere Energiebedarf bei statischen Variablen mit ihrer globalen Verfügbarkeit und der Art der Speicherung zu tun haben kann. Statische Variablen sollten demzufolge in nicht-statische Variablen umgewandelt werden. Dies birgt allerdings Konfliktpotenzial in Bezug auf den vierten Energy Smell von Palomba et al. (2019), der eine Umwandlung nicht-statischer Methoden in statische erfordert. Listing 6 veranschaulicht den Konflikt.

```
public class Calculator {

  public static int bonus = 100;

  public int addBonus(int base) {
    return base + bonus;
  }
}
```

Listing 6: Konfliktpotenzial in Bezug auf statische und nicht-statische Variablen und Methoden.

Die Variable `bonus` in Listing 6 ist statisch. Somit ist kein Objekt der Klasse `Calculator` für den Zugriff auf sie nötig. Bei Orientierung an den Ergebnissen von Kumar et al. (2017) sollte diese Variable in eine nicht-statische umgewandelt werden.

Die `addBonus`-Methode hingegen ist nicht-statisch und greift auf keine Instanzvariable zu (`bonus` ist schließlich statisch), daher sollte sie nach Palomba et al. (2019) in eine statische umgewandelt werden, damit kein `Calculator`-Objekt erzeugt werden muss, welches Energie und Speicher benötigen würde.

Beide Refactoring-Techniken anzuwenden, würde dafür sorgen, dass der Code nicht mehr kompiliert: Es ist nicht möglich, in einer statischen Methode nicht-statische Variablen ihrer Klasse aufzurufen. (Damit die Variable überhaupt verfügbar ist, ist eine Instanz vonnöten.) Aus logischen Gründen können also in diesem Fall nicht beide Techniken angewendet werden. Hinzu kommt der größere Kontext: Vielleicht gibt es außerhalb der Klasse `Calculator`

weitere Aufrufe von `bonus` oder `addBonus()`, die ihrerseits einen Einfluss auf den Energieverbrauch haben, wenn man sie ändert.

Um in so einem Fall eine Entscheidung zu treffen, die der Energieeffizienz zuträglich ist, müssen neben dem Kontext auch die Gründe berücksichtigt werden, aus denen die eine oder andere Technik angewendet werden soll. Statische Variablen gilt es laut Kumar et al. (2017) vor allem wegen ihrer Art der Speicherung zu vermeiden, die zu Einbußen bezüglich Energie und Zeit führt. Nicht-statische Methoden, die auch statisch sein könnten, erfordern unnötige Instanziierung von Objekten. Dies trifft aber auch auf nicht-statische Variablen zu, falls sie von außen aufgerufen werden.

Die Handlungsanweisung, statische durch nicht-statische Variablen zu ersetzen, wird trotz des oben dargelegten Konfliktpotenzials in die Liste der hier angewendeten Refactoring-Techniken aufgenommen, mit dem Vorbehalt, dass im Einzelfall der Kontext nicht außer Acht gelassen werden darf. Für die Fallstudien im Rahmen dieser Bachelorarbeit wird es zwei Versionen der Kategorie Energy Smells geben, die jeweils neben den anderen Refactoring-Techniken nur eine dieser beiden Maßnahmen umsetzt. Ausführliche Erläuterungen dazu folgen in Kapitel 4.

Laut den weiteren Ergebnissen ist der Umgang mit primitiven Datentypen wesentlich energie- und zeitsparender als der mit Wrapper-Typen. Kumar et al. (2017) empfehlen daher, immer primitive Datentypen zu bevorzugen, wenn bei der Programmierung die Energieeffizienz im Vordergrund steht. Jedoch sei hier auf die in Abschnitt 2.3.3 dargelegten Erkenntnisse von Hasan et al. (2016) verwiesen, die feststellten, dass das Speichern primitiver Datentypen in Datenstrukturen in Java mehr Energie benötigt als das Speichern ihrer Wrapper-Typen, da andernfalls ein Autoboxing stattfindet, das wiederum Energie verbraucht. Die Verwendung primitiver Datentypen ist demnach nur dann zu empfehlen, wenn sie nicht in Datenstrukturen gespeichert werden sollen.

Um die effizienteste Methode zur String-Konkatenation in Java zu ermitteln, haben Kumar et al. (2017) die benötigte Energie und Zeit bei hundertmaligem Anhängen einer Zahl an denselben String gemessen. Sie verwendeten dabei die `append`-Methoden von StringBuffer und StringBuilder sowie die Konkatenation mit dem +-Operator. Insgesamt zeigte sich, dass der +-Operator sowohl hinsichtlich der Zeit als auch der Energie die effizienteste Option war. Der

Verbrauch von StringBuilder war um 10% höher, während der von StringBuffer um 4% höher lag.

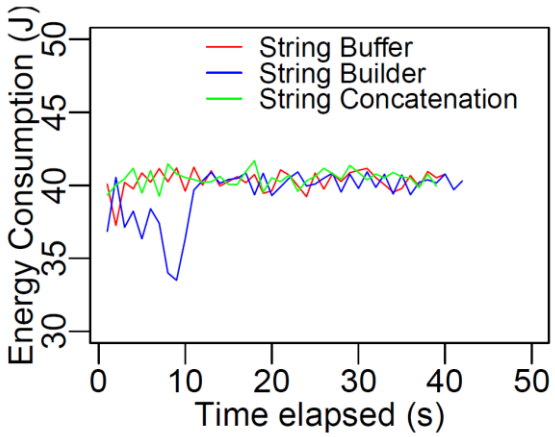


Abbildung 2: Energieverbrauchsmessung bei String-Konkatenation (Kumar et al., 2017).

In Abbildung 2 ist zu sehen, dass StringBuilder zunächst am wenigsten Energie verbrauchte. Da er die meiste Zeit benötigte, ist sein Gesamtverbrauch jedoch am höchsten. Die Autor:innen schließen an dieser Stelle, dass aufgrund des Gesamtverbrauchs der +-Operator die beste Wahl sei. Aus den Resultaten in Abbildung 2 ließe sich auch ableiten, dass bei einer geringen Anzahl von Konkatenationen StringBuilder zu bevorzugen sei. Allerdings erscheint es wahrscheinlich, dass eine Instanziierung eines neuen StringBuilder-Objekts bei wenigen Konkatenationen die Energieeffizienz eher negativ beeinträchtigen würde. Die Autor:innen äußern sich leider nicht weiter zur Bedeutung oder möglichen Interpretationen ihrer Resultate.

Basierend auf den Ergebnissen von Kumar et al. (2017) ergeben sich abschließend folgende Refactoring-Techniken:

- Vermeidung der Datentypen byte, short, float, double und char
- Umwandlung von statischen Variablen in nicht-statische (sofern es in dem gegebenen Kontext sinnvoll erscheint)
- Verwendung primitiver Datentypen (Ausnahme bei Speicherung in Datenstrukturen)
- Verwendung von + zur String-Konkatenation

Im Übrigen stellten Kumar et al. (2017) in ihrer Studie fest, dass IntelliJ IDEA von den einbezogenen Integrated Development Environments (IDEs) die energieeffizienteste ist. Eclipse und

NetBeans verbrauchten im Schnitt 10% und 12% mehr Energie. Im praktischen Teil dieser Bachelorarbeit wird IntelliJ IDEA verwendet.

2.4 Weitere Ansätze

Neben dem manuellen Refactoring, bei dem Entwickler:innen verbesserungsbedürftige Strukturen im Code identifizieren und umschreiben, wie es auch in dieser Arbeit geschieht, gibt es noch weitere Herangehensweisen, die zum Abschluss dieses Kapitels nicht unerwähnt bleiben sollen.

2.4.1 Automatisiertes und KI-unterstütztes Refactoring

Ein relativ junger Ansatz in der gegenwärtigen Literatur zu diesem Thema ist das automatisierte Refactoring. Cruz & Abreu (2019) sowie Bree & Cinnéide (2021) wählen diese Herangehensweise und schlagen Werkzeuge vor, die vorhandenen Programmiercode statisch analysieren und ein Refactoring durchführen sollen.

Zwar verfügen die meisten gängigen IDEs bereits über Tools oder Plugins zur Erkennung von Code Smells und auch zur Durchführung kleinerer Refactorings, jedoch fokussieren sich diese bislang fast ausschließlich auf traditionelle Code Smells im Sinne Fowlers. Neuere Ansätze sollen nun einerseits Refactorings zur Steigerung der Energieeffizienz durchführen und dies andererseits möglichst großflächig, selbstständig und zuverlässig bewerkstelligen.

Diesem Vorhaben stehen allerdings noch einige Hürden im Wege, die von Cruz & Abreu (2019) hervorgehoben werden: Zwar konnte ihr Programm einige Energy Smells ermitteln und beheben, und tatsächlich führte dies zu weniger Energieverbrauch. Allerdings identifizierte das Programm auch Energy Smells, die gar keine waren, und modifizierte den Code an diesen Stellen ungefragt. Die Refactorings führten teilweise zu weniger verständlichem Code, der für die Entwickler:innen, an deren Code das Programm erprobt wurde, schwer nachzuvollziehen war. Zudem war das Programm auf fünf Android-spezifische Code-Smells beschränkt, die sich teilweise auf konkrete Klassen beziehen. An dem letzten Punkt wird das größte Defizit an

automatischen Refactorings für mehr Energieeffizienz deutlich: Während es für klassisches Refactoring bereits feststehende und eindeutige Regeln gibt, ist der Bereich der Energy Smells möglicherweise noch nicht ausreichend erforscht, um verallgemeinernde Regeln anzuwenden. Im vorangegangenen Abschnitt wurde bei der Auflistung der Refactoring-Techniken bereits deutlich, dass es Widersprüche geben kann und nicht immer klar ist, welche Maßnahmen zu einer Steigerung oder Senkung des Energieverbrauchs führen werden. Ein automatisiertes Refactoring setzt voraus, dass das Programm, das es ausführen soll, entweder über klare Kriterien und Handlungsanweisungen verfügt – oder vorhersagen kann, welche Auswirkungen die Code-Änderungen auf die Energieeffizienz haben werden.

Um solche Vorhersagen zu treffen, gibt es erste Ansätze, die auf maschinelles Lernen zurückgreifen: Imran et al. (2023) machen es sich in einer aktuellen Arbeit zur Aufgabe, die Auswirkung von Refactorings auf die Energieeffizienz sowie den Speicherbedarf vorherzusagen. Dazu haben sie 16 Arten von Code Smells und ihren gemeinsamen Einfluss auf die Ressourcennutzung in 31 Open-Source-Anwendungen untersucht. Es erfolgte eine detaillierte empirische Analyse der Veränderung der CPU- und Speichernutzung nach der Einzel- und Gruppen-Refactoring bestimmter Code Smells. Basierend auf dieser Analyse trainierten Imran et al. (2023) Regressionsalgorithmen, um den Einfluss der Gruppen-Refactorings auf die CPU- und Speichernutzung vorhersagen zu können. Anhand solcher Vorhersagen, so die Autor:innen, sei es zukünftig möglich, den Effekt eines Refactorings auf die Energieeffizienz und Speicherbedarf vorherzusagen. Da auch dieser Forschungsbereich noch jung ist, ist zu erwarten, dass es in kommenden Jahren weitere Untersuchungen zu dieser Thematik geben wird.

Sowohl automatisierte als auch KI-gestützte Refactorings haben den klaren Nachteil, dass die Entwickler:innen selbst potenziell keine informierten Entscheidungen mehr treffen, sondern sich nur auf das jeweilige Programm verlassen könnten. Im Optimalfall jedoch werden bereits beim erstmaligen Schreiben jegliche Arten von Code Smells vermieden. Guten Code (beispielsweise im Sinne der Energieeffizienz) zu produzieren, setzt voraus, dass bekannt ist, wie solcher aussehen sollte. Diese Tatsache begründet die Relevanz der näheren Betrachtung von Refactorings mit dem Ziel, die Energieeffizienz zu steigern.

2.4.2 Profiler

Zur Performance-Optimierung werden von Entwickler:innen häufig sogenannte Profiler eingesetzt. Dabei handelt es sich um Tools, die es ermöglichen, interne Vorgänge einer Anwendung zu analysieren, wie beispielsweise die Speicherzuweisung, CPU-Auslastung oder Laufzeit. Namhafte Java-Profiling-Tools sind VisualVM, JProfiler oder YourKit. Je nach Tool lassen sich besonders ressourcenintensive Bereiche im Code identifizieren. Auch hier liegt der Fokus noch selten auf dem Energieverbrauch, wobei mit Blick auf die letzten Jahre durchaus Veränderungen zu verzeichnen sind.

Pereira et al. (2020) entwickelten beispielsweise eine Technik namens SPELL (SPectrum-based Energy Leak Localization), die sie in einem Tool implementierten, das durch Code-Analyse besonders ineffiziente Bereiche einer Anwendung identifiziert hervorhebt.

Sie ließen 15 Entwickler:innen eine Reihe von Java-Anwendungen Refactorings hinsichtlich der Energieeffizienz durchführen; dabei verließ sich eine Gruppe auf ihre eigenen Einschätzungen, eine zweite verwendete das SPELL-Tool, und eine dritte identifizierte performancekritische Code-Abschnitte mithilfe des Profiling-Tools der IDE NetBeans, das zwar die CPU-Auslastung, nicht aber den Energieverbrauch misst. Den Resultaten dieses Versuchs zufolge konnte das Refactoring, das sich auf SPELL stützte, mit durchschnittlichen 44% die höchste Steigerung der Energieeffizienz erreichen.

Nach Veröffentlichung der Arbeit wurde das Tool leider nicht weiterentwickelt. Somit ist es für es für Anwendungen, die neuere Java-Versionen verwenden, nicht brauchbar, da beispielsweise eine alte Version des JavaParsers verwendet wird, der manche Konstrukte, die ab Java 8 neu hinzukamen, nicht verarbeiten kann.

Weitere Profiling-Tools, die den Energiekonsum berücksichtigen, sind nur spärlich vorhanden und in der Regel noch nicht erprobt, was möglicherweise auch damit zusammenhängt, dass die Vermutung im Raum steht, performanter Code sei energieeffizienter Code, und somit würden die Profiler ausreichen, die Aufschluss über die Performance geben. Daher wird zur Energieverbrauchsmessung von Programmcode im wissenschaftlichen Bereich häufig auf Energie-Profiler zurückgegriffen, die den Energieverbrauch eines Computers messen und aufzeichnen, allerdings nicht für die Analyse einzelner Anwendungen optimiert sind. Hier kann sowohl auf

hardware- als auch auf softwarebasierte oder hybride Tools zurückgegriffen werden (Kruglov et al., 2023). In dieser Arbeit wird ein softwarebasiertes Tool für Geräte mit Intel-Prozessor namens Power Gadget verwendet (mehr dazu in Kapitel 4).

2.4.3 Empfehlungen

Da Softwareentwickler:innen häufig keinen Einfluss auf die genutzte Hardware der Anwender:innen haben, ist der Einfluss von Hardware-Komponenten auf die Energieeffizienz nicht Gegenstand dieser Arbeit. Fernab dessen gibt jedoch es einen weiteren wichtigen Faktor, der sich zwischen Code und Hardware befindet: die Java Virtual Machine (JVM).

Um Anwendungen auszuführen, die in Java geschrieben sind, ist eine entsprechende Laufzeitumgebung vonnöten: die Java Runtime Environment (JRE). Diese beinhaltet Librarys, eine JVM und andere für die Ausführung notwendige Komponenten. Um Java-Anwendungen zu entwickeln, muss darüber hinaus ein Java Development Kit (JDK) vorhanden sein (siehe Abbildung 3). Dieses enthält neben einer JRE unter anderem Compiler, Debugger und das Softwaredokumentationswerkzeug JavaDoc. (Programiz, 2023)

Bei der Ausführung einer Java-Anwendung wird der Java-Quellcode von der JRE in sogenannten Bytecode übersetzt. Dieser Bytecode ist eine Zwischensprache, die plattformunabhängig ist und von der auf dem jeweiligen Gerät installierten JVM interpretiert wird. Die JVM führt den Bytecode aus und wandelt ihn während der Laufzeit in Maschinencode um, der von der jeweiligen Hardware des Geräts ausgeführt werden kann. (Petri & Petri, 2023)

Wie bei der Hardware haben Softwareentwickler:innen normalerweise keinen direkten Einfluss darauf, welche JVM von Endanwender:innen verwendet wird. Die JVM wird in der Regel von ihnen selbst oder Systemadministrator:innen installiert und verwaltet. Als Softwareentwickler:in ist es jedoch durchaus möglich, bestimmte Anforderungen oder Empfehlungen hinsichtlich der verwendeten JVM in der Dokumentation oder den Systemanforderungen der Anwendung anzugeben.

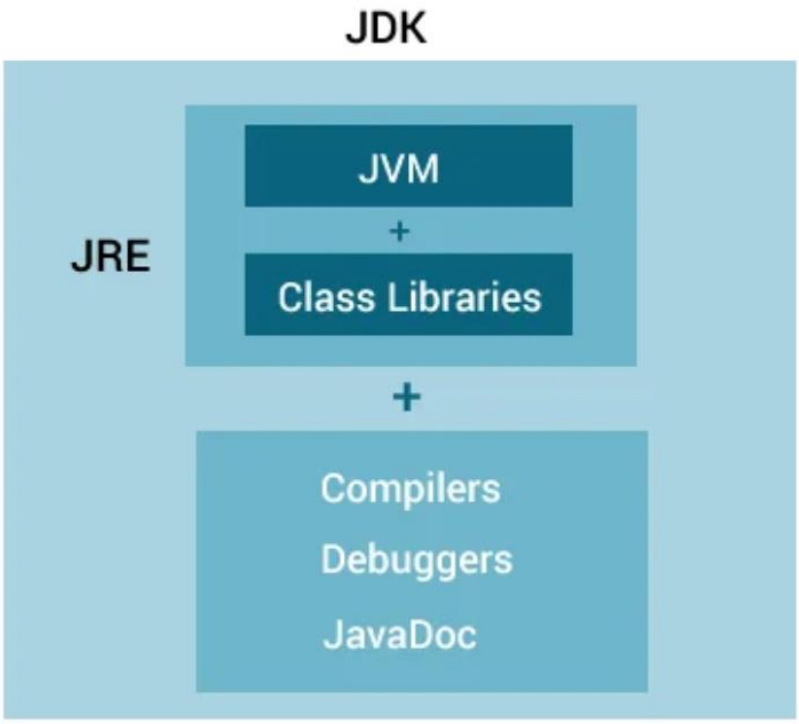


Abbildung 3: Verhältnis zwischen JDK, JRE, JVM und Compiler (Programiz, 2023).

Long et al. (2023) haben vier aktuelle JVM-Distributionen unter anderem hinsichtlich ihrer Energieeffizienz überprüft und verglichen: GraalVM von Oracle, Amazon Corretto, Adopt OpenJDK und Zulu. Insgesamt hat diese Studie deutlich gezeigt, dass GraalVM-basierte Java-11-Anwendungen in den meisten Fällen bessere Leistung erbringen, weniger Energie verbrauchen und eine geringere CO_2-Äquivalent-Belastung aufweisen als solche, die die anderen durchaus verbreiteten JVM-Distributionen nutzen. Daraus lässt sich ableiten, dass, auch wenn die Wahl der JVM-Distribution letztendlich allein in der Hand der Anwender:innen liegt, auch diese Entscheidung einen Einfluss auf die gesamte Energieeffizienz der Anwendung haben kann. Durch Empfehlungen kann diese Entscheidung möglicherweise in eine ressourcensparendere Richtung gelenkt werden.

Die Untersuchung des Energiesparpotenzials unterschiedlicher Komponenten des JDK wäre ein weiteres spannendes Thema, das jedoch weit über die Grenzen dieser Arbeit hinausreicht.

3 Fallstudien

Für die Durchführung der Fallstudienanalyse wurden zwei Java-Anwendungen ausgewählt, an denen die oben herausgearbeitete Refactoring-Techniken angewendet und deren Energieverbrauch in unterschiedlichen Code-Zuständen gemessen wird. Bei den ausgewählten Anwendungen handelt es sich um die in Java geschriebenen Open-Source-Projekte GraphStream und LanguageTool. Um eine geeignete Auswahl zu treffen und sicherzustellen, dass die analysierten Anwendungen repräsentative Ergebnisse liefern, wurden bestimmte Auswahlkriterien definiert.

3.1 Auswahlkriterien

Im Folgenden sind die Kriterien, auf denen die Auswahl basiert, aufgelistet und kurz erläutert. Anschließend werden die ausgewählten Anwendungen vorgestellt.

Implementierung in Java: Erstens ist es von Bedeutung, dass die ausgewählten Anwendungen in Java entwickelt wurden, da die Untersuchungen in dieser Arbeit auf der Analyse von Java-Anwendungen zur Verbesserung der Energieeffizienz basieren.

Open-Source: Zweitens sollten die ausgewählten Anwendungen open-source sein, um Zugang zum Quellcode zu erhalten und eine transparente Analyse zu ermöglichen.

Hoher Anteil an eigenem Code: Ein weiteres wichtiges Kriterium ist, dass die ausgewählten Anwendungen einen beträchtlichen Anteil an eigenem Code enthalten und nur wenig externen Code als Imports oder Dependencys verwenden. Dadurch wird gewährleistet, dass die

durchgeführten Refactoring-Maßnahmen und Messergebnisse hauptsächlich auf den internen Strukturen und Implementierungen der Anwendungen basieren und das Refactoring potenziell einen relativ größeren Effekt erzielt.

Angemessene Größe: Das Größenverhältnis der Anwendungen sollte angemessen sein, um eine effiziente Untersuchung zu ermöglichen. Aus diesem Grund werden Anwendungen ausgewählt, die zwar umfangreich genug sind, um signifikante Ergebnisse zu erzielen, jedoch nicht so umfangreich, dass sie den Rahmen dieser Arbeit sprengen würden. Ein Umfang zwischen 5.000 und 20.000 Zeilen Code erscheint sinnvoll.

Unit-Tests: Abschließend ist es für die Messungen unerlässlich, dass bereits Unit-Tests im Projekt vorhanden sind. Eine möglichst hohe Testabdeckung ist dabei wünschenswert, da hierdurch einerseits sichergestellt werden kann, dass durch das Refactoring die Funktionalität der Anwendung nicht verändert wird, und andererseits die Energieverbrauchsmessung auf den vorhandenen und zu ergänzenden Tests basiert (mehr dazu in Kapitel 4).

3.2 GraphStream

GraphStream ist eine open-source Java-Bibliothek, die Entwickler:innen eine Vielzahl von Funktionen und Algorithmen bietet, um komplexe graphenbasierte Strukturen zu erstellen und zu verändern, zu analysieren und darzustellen.

Die Bibliothek ermöglicht es, Graphen zu erstellen, Knoten und Kanten hinzuzufügen, Gewichtungen zu definieren und benutzerdefinierte Attribute für Knoten und Kanten zu setzen. Darüber hinaus bietet GraphStream leistungsstarke Algorithmen, beispielsweise zur Berechnung von kürzesten Pfaden, Fluss- und Netzwerkverbindungen oder Zentralitätsmaßen.

Dem Projekt GraphStream sind auf GitHub (einzusehen unter https://github.com/graphstream) mehrere Repositorys zugeordnet, die den Code für verschiedene Einsatzmöglichkeiten der Bibliothek bereitstellen. Das Repository mit dem Namen *gs-core* bietet alle nötigen Basisfunktionen. Es stellt die grundlegenden Klassen und Methoden für alle anderen Repositorys bereit, wie *gs-geography*, welches die Verwendung von Geodaten unterstützt, und *gs-algo*, in dem Implementierungen diverser Graphenalgorithmen zu finden sind. Da es sich bei den meisten

Repositorys um weitgehend unabhängige Module handelt, die sich lediglich auf *gs-core* als externe Dependency stützen, ist es für den Zweck dieser Fallstudienanalyse sinnvoll, das Refactoring auf eines dieser Module zu beschränken, um die Auswirkungen der Code-Änderungen in einem klar definierten Kontext zu analysieren und zu bewerten. Da ein Messvorgang jeweils die Ausführung aller Tests eines Moduls erfordert, wird eine Fallstudie nur ein Modul umfassen.

Für die Fallstudienanalyse wurde das Repository *gs-algo* als Ziel für das Refactoring ausgewählt, da es umfangreichen Code mit komplexer Logik und aufwendigen Berechnungen beinhaltet. Diese Komplexität bietet ein hohes Potenzial für mögliche Optimierungen durch das Refactoring, da effiziente Implementierungen hier von besonderer Bedeutung sind.

Das Modul beinhaltet 11 Packages (detaillierter einzusehen in der Dokumentation unter https://graphstream-project.org/gs-algo/), die unterschiedliche Arten von Algorithmen und verwandten Funktionalitäten bereitstellen. Es umfasst 6704 Zeilen Code.

Die Testabdeckung von *gs-algo* beträgt zum Startzeitpunkt der Untersuchungen bei Klassen 31%, bei Methoden 28% und bei Zeilen 33%. Für das Refactoring werden die nötigen Tests ergänzt, um einerseits sicherzustellen, dass das Verhalten des Codes nicht unbeabsichtigt verändert wird, und andererseits Messungen zu gewährleisten, bei denen alle Bereiche des Codes einbezogen werden, in denen das Refactoring durchgeführt wird.

Genaueres zu den Messungen und der Methodik folgt in Kapitel 4. Zuvor wird jedoch die zweite Anwendung vorgestellt, die Gegenstand des Refactorings und der Untersuchungen ist.

3.3 LanguageTool

LanguageTool ist ein Open-Source-Tool und eine Java-Bibliothek, die für die Grammatikprüfung, Rechtschreibprüfung und sprachliche Analyse von Texten entwickelt wurde. Es ermöglicht die automatische Überprüfung von Texten auf Fehler in Rechtschreibung, Grammatik, Interpunktion und Stil. LanguageTool kann in verschiedene Textverarbeitungsprogramme, Editoren und Plattformen integriert werden und bietet eine Vielzahl von Sprachunterstützungen.

Das Tool arbeitet regelbasiert und kann aufgrund seiner Erweiterbarkeit um eigene Regelsätze ergänzt werden. Es kann sowohl von Entwickler:innen als Bibliothek in ihre Anwendungen eingebettet als auch als eigenständige Anwendung genutzt werden, um Texte auf sprachliche Fehler zu überprüfen und Verbesserungsvorschläge zu machen. Der komplette Quellcode ist frei zugänglich unter https://github.com/languagetool-org/languagetool einsehbar.

Auch hier handelt es sich um ein Projekt, das in mehrere Module aufgeteilt ist, auch wenn sich diese im Falle von LanguageTool alle im selben Repository befinden. Dabei enthält das Modul *languagetool-core* fast die gesamte Logik und den größten Teil des Codes, der für die Kernfunktionalität für die grammatikalische Analyse und Fehlererkennung nötig ist.

Die anderen Module dienen spezifischen Zwecken, wie beispielsweise der Kommandozeilennutzung oder der HTTP-Kommunikation. Verglichen mit *languagetool-core* sind sie sehr klein und enthalten nur wenige Klassen, in denen sich zudem kaum Code findet, der sich für das Refactoring eignet.

Aufgrund des oben beschriebenen Aufbaus und um in einem einer Bachelorarbeit angemessenen Aufwandsrahmen zu bleiben, beschränkt sich das Refactoring von LanguageTool in dieser Arbeit auf den Unterordner *.rules* des Kernmoduls *languagetool-core*. Er umfasst 19.057 Zeilen Code. Die Tests in diesem Teil des Projekts decken 44% der Klassen, 26% der Methoden und 28% der Zeilen ab. Wie bei GraphStream werden auch hier vor Beginn der Messungen Tests ergänzt.

Die beiden betreffenden Repositorys wurden zwecks der mit dieser Bachelorarbeit verbundenen Änderungen mit dem Stand vom 03.08.2023 geforkt. Diese abgeleiteten eigenständigen Versionen und alle Änderungen an ihnen sind unter https://github.com/ninjajajaja/gs-algo-ee und https://github.com/ninjajajaja/languagetool-ee öffentlich einsehbar.

4 Methodik

Zur Messung des tatsächlichen Energieverbrauchs von Programmiercode muss dieser in irgendeiner Weise ausgeführt werden. Um die Ergebnisse dieser Messungen vergleichen und Zusammenhänge erkennen zu können, muss die Messung und damit auch die Ausführung des Codes unter reproduzierbaren Umständen erfolgen.

Unit-Tests erfüllen diese Bedingungen. Sie führen vollständig automatisiert alle Aufrufe und Instanziierungen aus, die zuvor in diesen Tests definiert wurden. Unit-Tests sind leicht erweiterbar. Um den Aufwand, der mit den Messungen in dieser Bachelorarbeit einhergeht, einzugrenzen, wurden daher Anwendungen gewählt, die bereits über Unit-Tests verfügen. So ist das Testing-Framework (hier bei beiden Anwendungen JUnit) bereits in das Projekt eingebunden und die Menge der übrigen zu schreibenden Tests ist geringer. Die gesamten Tests sind dann in einem einzigen Durchlauf ausführbar. So ein Durchlauf ist mit unterschiedlichen Zuständen des Codes (vor und nach einzelnen Refactorings) beliebig häufig wiederholbar.

Wie oben in Abschnitt 2.4.2 besprochen, sind Profiler mit Fokus auf Energie noch kaum verfügbar. Aus diesem Grunde wird hier, wie auch beispielsweise in der Studie von Şanlıalp et al. (2022), das hardwarebasierte Tool Intel Power Gadget verwendet, um Unterschiede im Energieverbrauch vor und nach Code-Änderungen zu messen. Der genaue Aufbau der Energieverbrauchsmessung in Kombination mit den Unit-Tests wird in Abschnitt 4.2 erläutert.

Im Folgenden werden der gesamte Ablauf und der Aufbau der Refactoring-Experimente erklärt.

4.1 Aufbau und Ablauf

Beide Fallstudien sind gleichermaßen aufgebaut. Der Aufbau beginnt mit der Erstellung von fünf Branches, die auf dem Ausgangszustand basieren. Diese Branches dienen dem Zweck, die Auswirkungen der Techniken in den Refactoring-Kategorien unabhängig voneinander messen zu können.

Es gibt je einen Branch für die Refactoring-Kategorien Design, Kontrollstrukturen und Datenstrukturen, und außerdem zwei für die Kategorie Energy Smells.

In der Kategorie Energy Smells gibt es zwei Refactoring-Techniken bezüglich (nicht-)statischer Methoden und Variablen, die sich potenziell widersprechen (auch noch einmal aufgeführt unter RNr. 13 und RNr. 17 in Tabelle 3 unten). Daher werden hier zwei Versionen erstellt, in denen neben den anderen Techniken der Kategorie Energy Smells je nur eine dieser beiden Techniken angewendet wird. Der einzige Unterschied zwischen den beiden Branches Energy Smells A und Energy Smells B wird also sein, dass entweder nicht-statische Methoden in statische oder statische Variablen in nicht-statische umgewandelt werden.

Auf jedem der fünf Branches werden nach deren Erstellung die jeweiligen Refactoring-Techniken der genannten Kategorien angewendet. Für alle Stellen im Code, die vom Refactoring betroffen und bisher noch nicht von Tests abgedeckt sind, werden auf dem Master-Branch Tests ergänzt, um sicherzustellen, dass alle Änderungen von den Messungen erfasst werden.

Sobald das Refactoring auf den fünf Branches und die Ergänzung aller entsprechenden Tests auf dem Master-Branch abgeschlossen sind, werden die durch die ergänzten Tests entstandenen Änderungen auf dem Master-Branch in die fünf Refactoring-Branches gemerget, um mit denselben Tests sowohl auf dem Master- als auch auf den Refactoring-Branches den Energieverbrauch zu messen. An manchen Stellen müssen auf den einzelnen Branches kleinere Anpassungen vorgenommen werden, damit das Projekt kompiliert: Wenn beispielsweise Getter entfernt wurden, werden die entsprechenden Aufrufe in den Tests durch den direkten Zugriff auf die Variable ersetzt. Abgesehen davon werden die Tests nicht mehr verändert, da dies die Vergleichbarkeit der Messergebnisse beeinträchtigen würde.

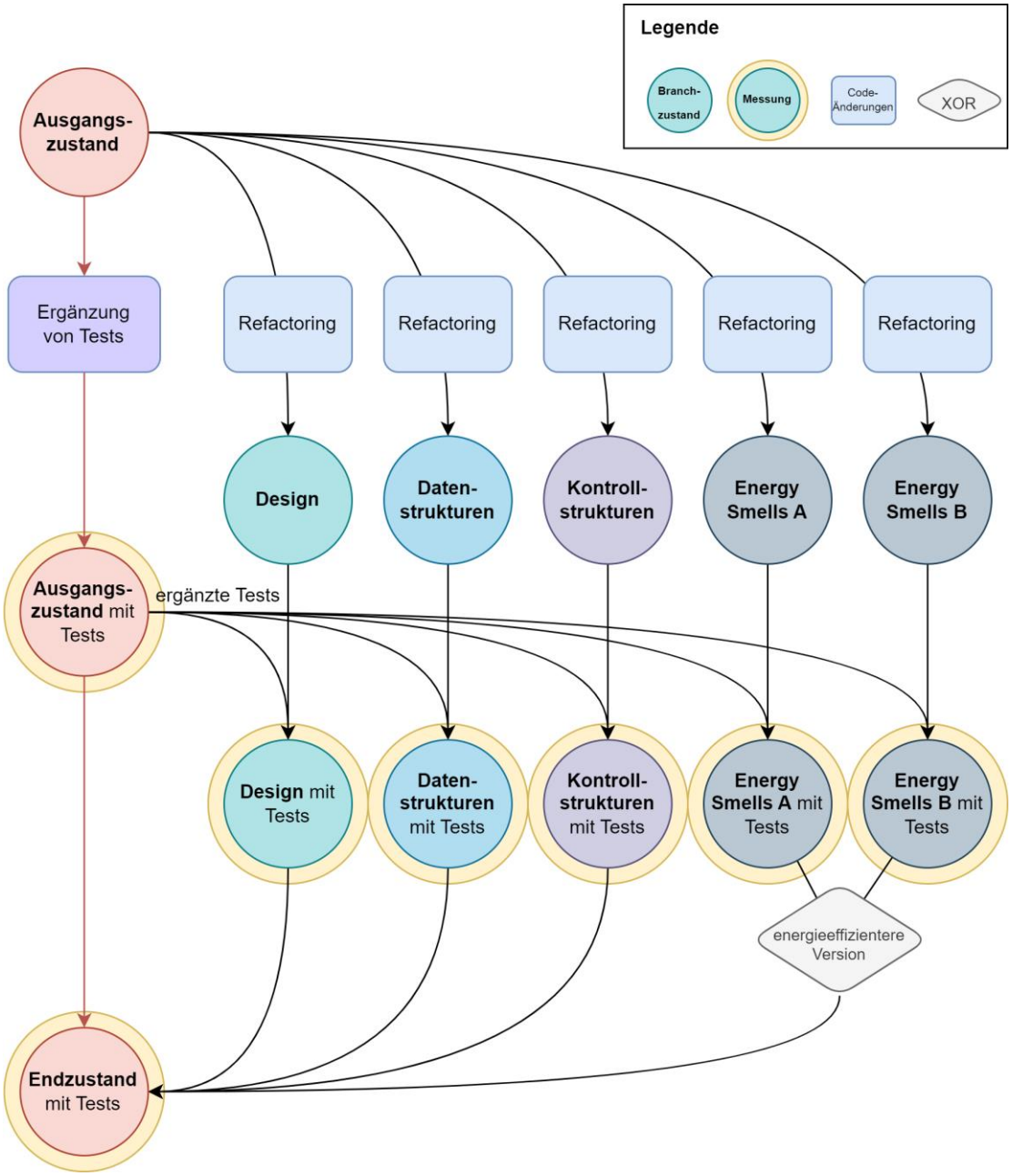


Abbildung 4: Ablauf des gesamten Refactoring-Vorgangs mit Messungen.

Nachdem alle Branches über dieselben Tests verfügen, wird der um die Tests ergänzte Ausgangszustand auf dem Master-Branch für die erste Messung herangezogen (in Abbildung 4 sind die zu messenden Zustände gelb umrandet). Gemessen und erhoben werden der Energie- und Zeitaufwand, die für das zehnmalige Ausführen aller Tests benötigt werden. Wie das genau

geschieht, ist in Abschnitt 4.2 ausführlich erklärt. Anschließend wird auf dieselbe Weise und unter denselben Bedingungen der Energieverbrauch der fünf Refactoring-Versionen gemessen.

Für die Messung des Endzustandes werden vier der fünf Branches wieder mit dem Master-Branch zusammengeführt. Von den beiden Energy-Smell-Branches A und B wird derjenige für die Zusammenführung ausgewählt, der eine höhere Energieeffizienzsteigerung aufweist.

Nach diesem abschließenden Merge werden die benötigte Zeit und Energie des Endzustandes gemessen.

4.2 Energieverbrauchmessung

Bisher scheint es kein standardisiertes Verfahren zur Messung des Energieverbrauchs oder der Performance für lokal ausgeführten Code zu geben. Die daraus resultierenden Unterschiede in der Methodik erschweren nicht nur die Vergleichbarkeit der Ergebnisse der in dieser Arbeit herangezogenen Quellen; sie beeinträchtigen auch die Wahl der geeigneten Methodik für die hier beschriebenen Experimente. Viele Quellen gehen nur oberflächlich auf den eigentlichen Messvorgang ein und präsentieren lediglich die Ergebnisse ihrer Messungen. Aus diesen Quellen geht nicht hervor, wie häufig sie die Messung unter welchen Bedingungen vorgenommen haben und wie der letztendlich genannte Wert ermittelt wurde.

Die vorhandenen Angaben, die sich auf die Häufigkeit der Messungen für einen Code-Zustand beziehen, schwanken etwas, lassen aber eine grobe Richtlinie erkennen: Imran et al. (2023) machten 7 Messdurchläufe und berechneten dann den Durchschnitt. Bei Palomba et al. (2019) und Şanlıalp et al. (2022) waren es 10, bei Bree & Cinnéide (2020) 25 und bei Tonini et al. (2013) 30 Durchläufe, bei denen der Energieverbrauch gemessen und der Durchschnitt berechnet wurde. Bree & Cinneide (2022) stechen mit einer Wiederholungsanzahl von 200 hervor, allerdings führten sie ihre Messungen an eher kurzen Code-Fragmenten durch, deren Ausführung nicht viel Zeit erfordert.

Da ein einmaliger Durchlauf aller Tests nur wenige Sekunden und Energie in Anspruch nimmt, werden die Tests je Messung zehnmal direkt nacheinander ausgeführt. Nähere Erläuterungen zur technischen Umsetzung in IntelliJ folgen in 4.2.2. Die Energieverbrauchsmessung je eines

zehnmaligen Testdurchlaufs wird in Anlehnung an die Methodik der Referenzliteratur zehnmal durchgeführt. Folgende Maßnahmen werden ergriffen, damit die Messungen unter möglichst ähnlichen Bedingungen stattfinden können:

- Vor jeder Messung wird der Projektcache bereinigt, die IDE neu gestartet und das Projekt neu gebaut.
- Der für die Messungen verwendete Laptop wird in den Flugmodus versetzt und befindet sich durchgehend im Netzbetrieb.
- Alle (sofern möglich auch die im Hintergrund laufenden) Anwendungen außer IntelliJ und Power Gadget werden geschlossen.
- Jegliche externen Geräte (wie zusätzliche Bildschirme) werden vom Laptop getrennt.

Wenn sich in einer Messreihe große Schwankungen in den Werten zeigen (mehr als 7% Unterschied zwischen der höchsten und der niedrigsten Messung) wird die Messreihe wiederholt, da die Messung von nicht zu unterbindenden Hintergrundprozessen beeinträchtigt worden sein kann.

Auf Basis der zehn Messungen pro Branch beziehungsweise Code-Zustand werden anschließend in Kapitel 5 Verfahren der deskriptiven Statistik angewendet, um eine aussagekräftige Interpretation der Ergebnisse zu ermöglichen.

Im Folgenden wird zunächst das Messwerkzeug vorgestellt, mit dem der Energieverbrauch ermittelt wird. Anschließend werden der technische Aufbau und die Integration in die Testumgebung in IntelliJ erklärt.

4.2.1 Messwerkzeug

Intel Power Gadget ist ein Tool, das von Intel entwickelt wurde und zur Messung des Energieverbrauchs von Computern mit Intel-Prozessor dient. Es ermöglicht die Überwachung des Stromverbrauchs von CPUs in Echtzeit und lässt sich kostenlos unter https://www.intel.com/content/www/us/en/developer/articles/tool/power-gadget.html herunterladen.

Bei Power Gadget lässt sich die Aufzeichnung der Messungen per Skript starten und stoppen. Dadurch kann die Energieverbrauchsmessung in die Testumgebung integriert und automatisiert werden, ohne sie manuell beginnen und beenden zu müssen. Dies ist entscheidend, um genaue und reproduzierbare Messergebnisse zu erhalten, da manuelles Starten und Stoppen zu ungenauen Ergebnissen und Zeitverzögerungen führen kann.

Zudem werden die Messergebnisse je eines Aufzeichnungsintervalls von Power Gadget in einem Logfile im csv-Format gespeichert. Diese Dateiformate lassen sich leicht zur weiteren Auswertung verarbeiten, um beispielsweise Durchschnittswerte mehrerer Messungen zu berechnen.

Aufgrund seiner oben genannten Eigenschaften und der Tatsache, dass sämtliche Messungen auf einem Gerät mit Intel-Prozessor und unter Windows 10 als Betriebssystem stattfinden, bietet das Intel Power Gadget eine effiziente und genaue Methode zur Energieverbrauchsmessung, die gut in die Testumgebung integriert und für die weitere Auswertung der Messergebnisse leicht zugänglich ist. Diese Eigenschaften machen es zu einem geeigneten Werkzeug für die Energieeffizienz-Analyse in der vorliegenden Arbeit.

4.2.2 Integration in die Testumgebung

IntelliJ bietet die Möglichkeit, Run-Konfigurationen zu erstellen. Diese dienen dazu, die Einstellungen und Parameter festzulegen, die für das Ausführen einer bestimmten Anwendung oder eines Projekts benötigt werden. Eine Run-Konfiguration definiert, wie der Code ausgeführt werden soll, welche Eingaben oder Argumente verwendet werden sollen, welche Umgebungsvariablen gesetzt werden müssen und welche Laufzeitoptionen erforderlich sind. Durch die Verwendung von Run-Konfigurationen kann der Ausführungsprozess für verschiedene Szenarien oder Anforderungen angepasst werden, ohne die Einstellungen jedes Mal manuell ändern zu müssen.

Innerhalb von Run-Konfigurationen gibt es die ‚Before launch'-Option. Diese ermöglicht es, vor Start der eigentlichen Anwendung Tests, Tasks, Build-Befehle, Skripte oder andere Anwendungen oder Run-Konfigurationen nacheinander auszuführen. Eine ‚After launch'-Option

gibt es bisweilen nicht. Ein Skript allein kann zudem nicht Basis einer Run-Konfiguration sein und kann lediglich als ‚Before launch'-Aufgabe eingestellt werden.

Aufgrund dieser Einschränkungen ist es nicht möglich, zuerst das Ausführen des Start-Skripts und anschließend das Ausführen der Tests als ‚Before launch'-Aufgaben für das Stopp-Skript einzustellen. Stattdessen wurde zum Zwecke der Messungen eine einfache Java-Klasse mit dem Namen AfterLogging erstellt, die als Basis einer Run-Konfiguration dient und lediglich den String „Finished!" auf der Konsole ausgibt. Die Run-Konfiguration dieser Klasse enthält drei ‚Before launch'-Aufgaben, sodass nacheinander das Start-Skript, die zehnmal die Tests und dann das Stopp-Skript ausgeführt werden, bevor letztendlich „Finished!" auf der Konsole ausgegeben wird.

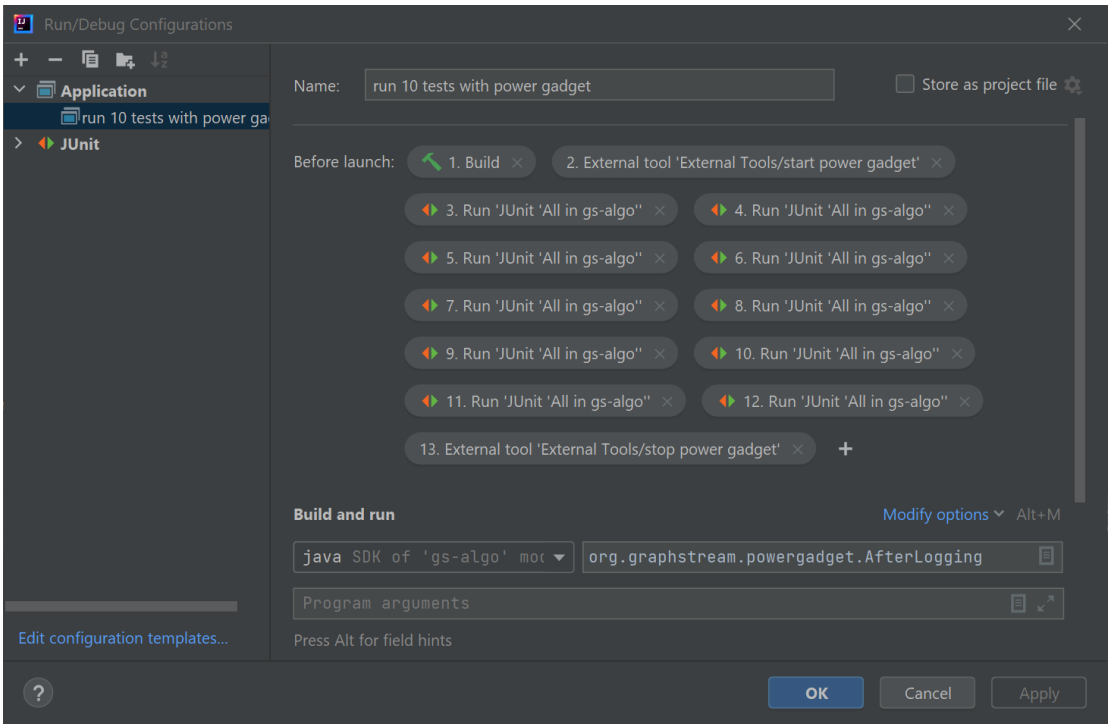


Abbildung 5: Run-Konfiguration zur Messung des Energieverbrauchs mit Power Gadget für *gs-algo*.

Abbildung 5 zeigt die Run-Konfiguration für *gs-algo*. Bei *languagetool-core.rules* wurden Einstellungen analog zu diesen verwendet.

Dieser Aufbau ermöglicht es, die Messung des Energieverbrauchs bei zehnmaligem Ausführen der Tests beliebig häufig unter reproduzierbaren Umständen zu wiederholen, um eine möglichst hohe Zuverlässigkeit und Vergleichbarkeit zu gewährleisten.

4.3 Refactoring und Testergänzung

Dieser Abschnitt beschreibt das methodische und praktische Vorgehen beim Refactoring der ausgewählten Anwendungen GraphStream und LanguageTool. Da die Ergänzung der notwendigen Tests parallel zum Refactoring stattfindet, ist dieser Schritt ebenfalls in diesem Unterkapitel dokumentiert.

Eine Übersicht der in Kapitel 2.3 erarbeiteten Refactoring-Techniken ist in Tabelle 3 dargestellt. Weiterhin folgen Erläuterungen zum Vorgehen zur Identifikation der Code-Abschnitte, die im Sinne dieser Regeln (RNr. 1 bis RNr. 19) modifiziert werden sollen. Besonderheiten und konkrete Zahlen zum Refactoring der einzelnen Anwendungen werden anschließend in separaten Abschnitten behandelt.

Regel-Nr.	**Technik**	**Beschreibung**
	Design	
RNr. 1	Reduzierung von Objekten	Wird eines der als ineffizient eingestuften Design-Patterns (Decorator, Abstract Factory oder Observer) identifiziert, wird die Anzahl der Objekte reduziert, indem das Pattern aufgelöst wird. **Quellen:** *Georgiou et al. (2020), Bree & Cinnéide (2022)*
RNr. 2	Reduzierung von Funktionsaufrufen	Wird eines der als ineffizient eingestuften Design-Patterns (Decorator, Abstract Factory oder Observer) identifiziert, wird die Anzahl der Funktionsaufrufe reduziert, indem das Pattern aufgelöst wird. **Quellen:** *Georgiou et al. (2020), Bree & Cinnéide (2022)*

Regel-Nr.	Technik	Beschreibung
RNr. 3	Ersetzen von Delegation durch Vererbung	Werden Aufrufe auf einem Objekt an ein Objekt einer anderen Klasse delegiert, sollte diese Delegation, wenn möglich, durch Vererbung ersetzt werden. **Quellen:** *Bree & Cinnéide (2020)*
RNr. 4	Entfernen von Gettern und Settern	Gettern und Setter werden entfernt und ein direkter Zugriff auf die jeweilige Variable ermöglicht. **Quellen:** *Tonini et al. (2013)*
	Kontrollstrukturen	
RNr. 5	Verwendung von for-each-Konstrukt bei allen Collections außer ArrayList	Bei for-Schleifen, innerhalb derer die Index-Variable nicht verwendet wird, wird das for-each-Konstrukt verwendet (es sei denn, es wird über eine ArrayList iteriert). **Quellen:** *Tonini et al. (2013)*
RNr. 6	Verwendung von for-Schleife mit Länge in externer Variable bei ArrayList	Bei Iteration über ArrayLists werden for-Schleifen mit Auslagerung der Größe in eine eigene Variable verwendet. **Quellen:** *Palomba et al. (2019)*
RNr. 7	Vereinfachung verschachtelter Schleifen	Schleifenverschachtelungen werden aufgelöst, sofern die Logik es zulässt. **Quellen:** *Kim et al. (2018), Şanlıalp et al. (2022)*
RNr. 8	Anpassung der Reihenfolge innerhalb von Bedingungen	Die Reihenfolge der einzelnen Teile einer Bedingung wird verändert, sodass der wahrscheinlichste Fall frühzeitig die gesamte Bedingung entscheidet, ohne die Logik zu verändern. **Quellen:** *Kumar et al. (2017)*
	Datenstrukturen	
RNr. 9	Kontextabhängiger Austausch von Collection-Datenstrukturen	Orientierung an Matrix (a), (b) und (c) nach Hasan et al. (2016), mit der Ausnahme, Hashtable statt HashMap zu verwenden. **Quellen:** *Hasan et al. (2016), Pereira et al. (2016)*
RNr. 10	Austausch von Datenstrukturimplementierungen	Hier ebenfalls Orientierung an Matrix (a), (b) und (c) nach Hasan et al. (2016). Neue Imports können nötig sein. **Quellen:** *Hasan et al. (2016)*

Regel-Nr.	Technik	Beschreibung
RNr. 11	Ersetzen nebenläufiger Datenstrukturen durch nicht-nebenläufige	Wenn Nebenläufigkeit nicht notwendig ist, werden nebenläufige Datenstrukturen durch ihr nicht-nebenläufiges Gegenstück ersetzt. **Quellen:** *Pereira et al. (2016)*
	(Weitere) Energy Smells	
RNr. 12	Überprüfen (und ggf. Beenden) von Threads, die bisher nicht explizit beendet werden	In einer angemessenen Frequenz wird überprüft, ob ein Thread noch läuft. Je nachdem, ob dies erwünscht ist, wird er beendet. **Quellen:** *Palomba et al. (2019)*
RNr. 13	Umwandlung nicht-statischer Methoden in statische, wenn sie auf keine Instanzvariablen ihrer Klasse zugreifen	Dem Methodenkopf wird der Zugriffsmodifikator 'static' hinzugefügt. **Quellen:** *Palomba et al. (2019)*
RNr. 14	Entfernen von ‚totem' (ungenutztem) Code	Code, der niemals aufgerufen wird, wird gelöscht. **Quellen:** *Hassan et al. (2017)*
RNr. 15	Speichern von Berechnungsergebnissen, die häufiger als einmal verwendet werden, in Variablen	Der Wert wird bei erstmaliger Berechnung in einer Variable gespeichert. Alle folgenden Berechnungen dieses Wertes werden durch Abfragen der entsprechenden Variable ersetzt. **Quellen:** *Hassan et al. (2017)*
RNr. 16	Vermeidung der Datentypen byte, short, float, double und char	Bei Zahlendarstellungen, die es erlauben, werden int oder long verwendet. **Quellen:** *Kumar et al. (2017)*
RNr. 17	Umwandlung von statischen Variablen in nicht-statische	Unter Berücksichtigung des Kontextes werden statische in nicht-statische Variablen durch Entfernen des Zugriffsmodifikators 'static' umgewandelt. **Quellen:** *Kumar et al. (2017)*
RNr. 18	Verwendung primitiver Datentypen (Ausnahme bei Speicherung in Datenstrukturen)	Wrapper-Typen werden durch die ihnen entsprechenden primitiven Datentypen ersetzt, sofern sie nicht in Collections gespeichert werden. **Quellen:** *Kumar et al. (2017)*
RNr. 19	Verwendung von + zur String-Konkatenation	Verwendung von StringBuilder oder StringBuffer zur Konkatenation wird durch den +-Operator ersetzt. **Quellen:** *Kumar et al. (2017)*

Tabelle 3: Übersicht über die 19 anzuwendenden Refactoring-Techniken.

Um die Stellen im Code zu identifizieren, die für das jeweilige Refactoring infrage kommen, wird hauptsächlich über die ‚Find in files'-Option in IntelliJ der Source-Ordner des Projekts rekursiv durchsucht. Die Tests werden bei der Suche nicht berücksichtigt. IntelliJ bietet die Möglichkeit, in dieser Suchoption reguläre Ausdrücke zu verwenden, was ein gezieltes Vorgehen überhaupt erst ermöglicht. Die verwendeten regulären Ausdrücke je RNr. sind im Anhang unter A.1 aufgeführt.

Es gibt drei Ausnahmen, bei denen die Suchfunktion nicht zur Ermittlung von Code-Stellen für das Refactoring verwendet werden kann: RNr. 13 (Umwandlung nicht-statischer Methoden in statische), RNr. 14 (Entfernen von totem Code) und RNr. 15 (Speicherung von Ergebnissen wiederholter Berechnungen in Variablen).

Für RNr. 13 und 14 wird daher das IntelliJ-Tool 'Code Inspection' verwendet. Bei der Ermittlung von totem Code wird berücksichtigt, dass Methoden, die zum Zeitpunkt des Refactorings nicht genutzt werden, potenziell von den Tests aufgerufen werden können, die erst im Laufe aller Refactorings ergänzt werden.

Da sich wiederholte Berechnungen desselben Werts innerhalb desselben Scopes weder per Regex noch mit der Code-Inspection-Funktion in IntelliJ ermitteln lassen, wird während des übrigen Refactorings darauf geachtet, ob solche Strukturen auffallen. Wenn dem so ist, werden sie für RNr. 15 vorgemerkt.

4.3.1 GraphStream

Refactoring

Im Zuge des Refactorings von GraphStream wurden (ohne Tests) 105 Dateien angepasst. Im Folgenden soll kurz auf die Besonderheiten in den einzelnen Kategorien eingegangen werden.

Design

RNr. 1 und 2: Das Projekt *gs-algo* enthält zwar eine abstrakte Generator-Klasse zur Generierung von Graphen, aber es handelt sich um keine Abstract Factory, da keine unterschiedlichen Ausprägungen eines abstrakten Datentyps erzeugt werden. Daher wurden für RNr. 1 und RNr. 2 keine Änderungen vorgenommen.

RNr. 3: Bezüglich Delegation gab es keine Vorkommen in der Hinsicht, dass eine Klasse nach dem Refactoring von einer anderen erbt, jedoch gab es einige mit @Override annotierte Methoden, die lediglich die gleichnamige Methode der Superklasse aufrufen. Diese wurden in insgesamt 6 Dateien entfernt.

RNr. 4: In der Kategorie Design wurden vor allem Getter und Setter entfernt, sofern sie lediglich der Zuweisung oder Abfrage von Variablen dienen. Die zugehörigen Variablen wurden, wenn nötig, mit dem Modifikator ‚public' versehen, um weiterhin den Zugriff von außen zu ermöglichen. Hierfür mussten 37 Dateien angepasst werden.

Kontrollstrukturen

RNr. 5 und 6: Zur Verbesserung von for-Schleifen im Sinne von RNr. 5 und 6 bietet *gs-algo* wenige Gelegenheiten, da häufig Streams (keine Collections) verwendet werden oder die Index-Variablen eine wichtige Rolle für die Logik spielen. Wenn möglich, wurde an solchen Stellen zumindest die size()-Abfrage aus der for-Zeile extrahiert und die Länge darüber in eigener Variable gespeichert, wie sonst nur für ArrayList vorgesehen. Wenn es mehrmals vorkam, wurde das für RNr. 16 (mehrfache Berechnung) vorgemerkt. Für RNr. 5 und 6 wurden 15 Dateien verändert.

RNr. 7: Das Projekt enthält nur sehr wenige verschachtelte Schleifen, und keine davon ist vereinfachbar, da auch hier die Index-Variablen eine wichtige Rolle spielen und beispielsweise mit Matrizen gerechnet wird.

RNr. 8: Bezüglich der Reihenfolge innerhalb von Bedingungen waren nur wenige Stellen eindeutig identifizierbar. Einige if-Verschachtelungen konnten aufgelöst werden, indem sie durch eine einzige durch && verbundene if-Abfrage ersetzt wurden. Außerdem wurden if-Abfragen,

die stets den Wert true haben, entfernt. Bei &&-Verknüpfungen wurde die aufwendigere Berechnung an die zweite Stelle verschoben, wenn beide mit gleicher Wahrscheinlichkeit true sind. Letztlich wurden, wenn möglich, Bedingungen vereinfacht. So wurde beispielsweise die Bedingung `((alive && !nalive) || (!alive && nalive))` verändert zu `(alive != nalive))`. Insgesamt wurden 10 Dateien modifiziert.

Datenstrukturen

RNr. 9 und 10: Unter Einbeziehung der in Abbildung 1 dargestellten Ergebnisse von Hasan et al. (2016) und denen von Pereira et al. (2016) wurden die Datenstrukturen und Implementierungen anhand folgender Richtlinien ersetzt:

Bei Listen: Da die Listen-Implementierungen von Trove nur für Datentypen, nicht aber Objekte verwendet werden können, wird bei Listen mit Datentypen immer Trove eingesetzt. Bei Objekt-Listen wird bei häufigem Einfügen am Ende auf ArrayList und bei häufigem Einfügen am Anfang auf LinkedList zurückgegriffen.

Bei Maps: Alle Maps werden durch Hashtable ausgetauscht.

Bei Sets: Alle Sets werden durch Trove-HashSets (mit <E> oder entsprechendem Datentypen) ersetzt, außer, wenn mit forEach() über das Set iteriert wird (Trove-Sets haben kein forEach() sondern nur Iterator, der wiederum zunächst aufgerufen werden müsste) - dann Hashtable wird verwendet.

RNr. 11: Während für RNr. 9 und 10 anhand dieser Richtlinien 30 Dateien angepasst werden mussten, gab es bezüglich RNr. 11 gar keine Vorkommen nebenläufiger Datenstrukturen.

Energy Smells A und B

Da sich die Refactorings für die letzten beiden Kategorien größtenteils überschneiden, werden sie in diesem Abschnitt gemeinsam besprochen.

RNr. 12: In dem Projekt werden nur an einer Stelle Threads verwendet. Diese werden allerdings wenige Zeilen später in derselben Methode per join() wieder zusammengeführt. Somit gab es hier keinen Refactoring-Bedarf.

RNr. 13: Nicht-statische Methoden, die auf keine Instanzvariablen zugreifen, wurden nur auf dem Branch der Kategorie ‚Energy Smells A' in 15 Dateien mit dem Zugriffsmodifikator ‚static' versehen. Hier handelte es sich meist um toString-Methoden für Knoten- und Kanten-Ids oder Methoden, die Fehler werfen. Trotz der sich hier aufdrängenden Frage nach der Sinnhaftigkeit dieser Refactoring-Maßnahme wurden die Modifikationen durchgeführt, um einen eventuellen Effekt auf den Energieverbrauch messen zu können.

RNr. 14: Beim Entfernen von totem Code wurden abgesehen von ungenutzten Methoden, Variablen und Parametern auch throws-Deklarationen an Methoden entfernt, in denen der betreffende Fehler niemals geworfen wird. Code, der von Library-Nutzenden durchaus von Relevanz sein könnte, wurde nicht gelöscht, auch wenn er nicht in den Tests aufgerufen wird. Dieses Refactoring betraf 16 Dateien, wobei drei Klassen komplett gelöscht wurden.

RNr. 15: Für das mehrmalige Berechnen desselben Werts konnten im Laufe des übrigen Refactorings einige Stellen in insgesamt 11 Dateien vorgemerkt werden, vor allem innerhalb von Schleifen.

RNr. 16: Der primitive Datentyp double kommt zwar sehr häufig vor, konnte aber nur in 4 Dateien durch int ersetzt werden, da in den meisten Fällen laut Spezifikation Fließkommazahlen berechnet werden.

RNr. 17: Dieses Refactoring wurde nur auf dem Branch der Kategorie ‚Energy Smells B' in 32 Dateien angewendet. In 30 davon wurde ein Objekt statisch initialisiert, um im Super-Konstruktor verwendet zu werden. Diese Initialisierung wird nun direkt im Super-Aufruf ausgeführt (siehe Listing 7). Es fällt also nicht nur die Eigenschaft dieser Objekte, statisch zu sein, weg, sondern auch die Zuweisung zu einer Variable.

```
// vor dem Refactoring
public class DyckGraphGenerator extends LCFGenerator {

	public static final LCF DYCK_GRAPH_LCF = new LCF(8, 5, -5, 13, -13);

	public DyckGraphGenerator() {
		super(DYCK_GRAPH_LCF, 32, false);
	}
}

// nach dem Refactoring
public class DyckGraphGenerator extends LCFGenerator {

	public DyckGraphGenerator() {
		super(new LCF(8, 5, -5, 13, -13), 32, false);
	}
}
```

Listing 7: Die Klasse DyckGraphGenerator vor und nach dem Refactoring. (Repräsentatives Beispiel für 30 Klassen mit derselben Struktur.)

RNr. 18: GraphStream bietet die Möglichkeit, Objekte als Attribute von Kanten und Knoten zu speichern. Um beispielsweise einen Double-Wert als Gewichtsattribut einer Kante zu speichern, muss diese Zahl also dennoch in seinen Wrapper-Typen ‚verpackt' werden. Aus diesem Grunde werden im Projekt *gs-algo* zwar häufig Wrapper-Typen verwendet, jedoch ist es fast nie (lediglich in zwei Dateien) sinnvoll oder möglich, sie durch ihre primitive Datentypenvariante zu ersetzen.

RNr. 19: Die String-Konkatenation betreffend gab es nur eine einzige Stelle im Code, an der StringBuilder verwendet und im Rahmen des Refactorings die Konkatenation durch Verwendung des +-Operators ersetzt wurde.

Tests

Das Projekt *gs-algo* bietet einige Graph-Generator-Klassen, die eine Instanz der Klasse Random verwenden. In diesen Klassen musste die Implementierung geringfügig geändert und ein

Seed gesetzt werden, um Schwankungen in den Messungen vorzubeugen. Außerdem wurde in der Klasse URLGenerator eine Methode, die auf das Internet zugreift, auskommentiert, um auch hier unnötige Schwankungen zu vermeiden.

Durch die Änderungen und Ergänzungen stieg die Anzahl der Tests in *gs-algo* von 72 auf 159. Die Testabdeckung änderte sich bei Klassen von 31% auf 65%, bei Methoden von 28% auf 58% und bei Zeilen von 33% auf 63%.

4.3.2 LanguageTool

Refactoring

Im Modul *languagetool-core* wurden im Laufe des Refactorings 146 Dateien angepasst (Tests ausgenommen). Klassen, die zur Serverkommunikation verwendet werden, wurden aus dem Ordner entfernt, da sie im Rahmen der Tests für die Messungen nicht ausgeführt werden würden. Wie im obigen Abschnitt zu GraphStream folgen kurze Bemerkungen zum konkreten Einsatz der einzelnen Refactoring-Techniken.

Design

RNr. 1 und 2: In dem in dieser Bachelorarbeit behandelten Scope von LanguageTool fand keines der Design-Patterns Observer, Abstract Factory oder Decorator Anwendung. Daher erfolgten hier keine Code-Änderungen.

RNr. 3: Es gibt eine Klasse namens EditOperation, in der Delegation an eine Instanz der Klasse Random durch Vererbung ersetzt werden könnte. Allerdings implementiert diese Klasse zusätzlich ein Interface, weshalb bei Vererbung an aufrufender Stelle ein mehrfaches Casten nötig wäre. Daher wurde die Delegation hier nicht durch Vererbung ersetzt. Weitere Vorkommen gibt es nicht.

RNr. 4: Mit @Override gekennzeichnete Getter und Setter aus abstrakten Klassen wurden nicht entfernt, da der Zugriff in den implementierenden Klassen sonst nicht gewährleistet werden kann. Abgesehen davon wurden Getter und Setter in 61 Dateien entfernt.

Kontrollstrukturen

RNr. 5 und 6: Bezüglich der effizienteren Schleifeniteration gab es kaum Vorkommen, bei denen die Index-Variable nicht für den Code-Block innerhalb der Schleife notwendig ist. Wenn möglich, wurde zumindest die size()-Abfrage aus der for-Zeile geholt und die Länge darüber in eigener Variable gespeichert, wie sonst nur für ArrayList vorgesehen.

RNr. 7: Keine der verschachtelten Schleifen im betrachteten Scope ist vereinfachbar.

RNr. 8: Ähnlich wie bei GraphStream konnten nur wenige Bedingungen identifiziert werden, innerhalb derer ein Tausch in der Reihenfolge sinnvoll erscheint. Auch hier wurden aber if-Abfragen, die immer den Wert true ergeben, entfernt, und Bedingungen vereinfacht, wo es möglich war. Insgesamt wurden in dieser Kategorie 49 Dateien modifiziert.

Datenstrukturen

RNr. 9 und 10: An zwei Stellen konnten ArrayList-Implementierungen, an denen immer an erster Stelle eingefügt wurde, durch LinkedList ersetzt werden. Andersheum wurden auch einige LinkedList-Verwendungen durch ArrayList ersetzt. Abgesehen davon wurden einige ArrayList-Objekte mit primitiven Datentypen durch ihre Trove-Version, Maps (über 200 Vorkommen) durch Hashtable und Sets durch die jeweilige Trove-Version ersetzt.

RNr. 11: Es gab einige Vorkommen von ConcurrentHashMap, die durch Hashtable ersetzt werden konnten. Von RNr. 9, 10 und 11 waren insgesamt 80 Dateien betroffen.

Energy Smells A und B

RNr. 12: Im betrachteten Scope gibt es keine Threads, die nicht explizit beendet werden.

RNr. 13: Dieses Refactoring betraf 91 Dateien, wobei 88 Methoden den Modifier 'static' bekamen. Zwar konnten einige nicht-statische Methodenaufrufe durch statische ersetzt werden, allerdings erforderte der Kontext häufig ohnehin ein Objekt der betreffenden Klasse, sodass trotz eines vorhandenen Objekts im selben Scope die statische Methode der Klasse aufgerufen wurde.

RNr. 14: Die Funktion 'Code Inspection' konnte zwar keine Methoden oder Variablen identifizieren, die nicht aus einem der anderen Module von LanguageTool heraus genutzt werden. Allerdings wurden in 50 Dateien redundante throws-Deklarationen entfernt, die meisten davon an Konstruktoren.

RNr. 15: Bezüglich unnötiger Mehrfachberechnungen gab mehrere Vorkommen in 27 Dateien, auch hier meist innerhalb von Schleifen. Beispielsweise wurde häufig innerhalb des Scopes einer for-Schleife mit Index mehrfach auf tokens[i] zugegriffen, anstatt diesen Wert einer Variable zuzuweisen. Dies wurde behoben.

RNr. 16: Da es sich um eine Anwendung zur Textverarbeitung handelt, ist häufig der Gebrauch von char unvermeidlich. Auch die anderen primitiven Datentypen werden sinnvoll verwendet (wie zum Beispiel double für Wahrscheinlichkeiten). Somit gab es hier keine Notwendigkeit, RN. 16 anzuwenden.

RNr. 17: In 54 Dateien konnte bei über 100 statischen Feldern der ‚static'-Modifikator entfernt werden, da es keine statischen Zugriffe auf diese Felder gab.

RNr. 18: Es konnten lediglich vier Stellen ermittelt werden, an denen primitive Datentypen statt ihrer Wrapper verwendet werden konnten, da diese in den übrigen Fällen entweder in Collections gespeichert wurden oder die Programmlogik es erfordert, dass sie ‚null' sein können.

RNr. 19: StringBuilder wird in diesem Projekt an diversen Stellen verwendet. In 12 Dateien konnten mehrere StringBuilder-Objekte durch String-Objekte ausgetauscht und die append-Aufrufe durch die entsprechende Konkatenation mit + oder += ersetzt werden. Lediglich in Kontexten, in denen weitere von StringBuilder angebotene Methoden wie replaceAll oder deleteCharAt verwendet wurden, wurde StringBuilder beibehalten. StringBuffer wird im hier betrachteten Scope nicht verwendet.

Tests

Aufgrund zahlreicher Abhängigkeiten und Methoden mit einer Länge von über 200 Zeilen erwies sich das Testen als wesentlich aufwendiger als bei GraphStream. Bei manchen sehr langen Methoden wurden daher nicht vom Refactoring betroffene Teile auskommentiert oder die modifizierten Stellen in eigene Methoden ausgelagert, die dann separat getestet werden konnten.

Durch die Änderungen und Ergänzungen stieg die Anzahl der Tests für den Ordner *.rules* in *languagetool-core* von 164 auf 212. Die Testabdeckung änderte sich bei Klassen von 44% auf 74%, bei Methoden von 26% auf 61% und bei Zeilen von 28% auf 57%.

4.3.3 Zusammenführen der Refactoring-Branches

Bei GraphStream traten keine Konflikte zwischen RNr. 13 und RNr. 17 (also zwischen Energy Smells A und B) auf. Daher wurde der energieeffizientere Energy-Smells-Branch für den ersten Endzustand ausgewählt und außerdem ein zweiter Endzustand erzeugt, der auch den weniger effizienten Energy-Smells-Branch (und somit sämtliche Refactorings aller Kategorien) beinhaltete.

Bei LanguageTool hingegen gab es an mehreren Stellen Konflikte zwischen den beiden Refactorings, sodass der vorgegebene Aufbau (siehe Abbildung 4) beibehalten und nur ein Endzustand erzeugt wurde, der neben den Änderungen der Branches für Design, Kontrollstrukturen und Datenstrukturen lediglich die des Energy-Smell-Branches enthält, der den Messergebnissen zufolge weniger Energie benötigt.

Dies führt dazu, dass es bei GraphStream acht und bei LanguageTool sieben gemessene Code-Zustände gibt.

5 Ergebnisse und Bewertung

In den vergangenen Kapiteln wurden auf Basis der bestehenden Forschungsliteratur auf Energieeffizienz abzielende Refactoring-Maßnahmen ermittelt, zwei Anwendungen für Fallstudien ausgewählt und der methodische Aufbau dargelegt. In diesem Kapitel werden die gemessenen Werte zunächst als Rohdaten präsentiert, anschließend statistisch analysiert und ausgewertet und schließlich die Ergebnisse hinsichtlich der Effektivität der Refactorings und des Zusammenhangs zwischen Energieeffizienz und Performance interpretiert. Zudem werden mit Blick auf den gesamten Versuchsaufbau kritische Aspekte und mögliche Einflussfaktoren diskutiert.

Bei den beiden Anwendungen GraphStream und LanguageTool stand jeweils ein Untermodul beziehungsweise -ordner im Fokus der Fallstudien. Im Rahmen dieser Arbeit wurden die zuvor herausgearbeiteten Refactoring-Techniken in fünf Kategorien eingeteilt: Design, Kontrollstrukutren, Datenstrukturen, Energy Smells A und Energy Smells B. Die letzten beiden unterscheiden sich lediglich in einer Refactoring-Maßnahme, die allerdings in Bezug auf die jeweils andere Konfliktpotenzial birgt. Die Refactorings einer Kategorie wurden jeweils auf einem Branch durchgeführt. Die Messungen erfolgten bei je zehnmaliger Ausführung der Unit-Tests für das untersuchte Modul (*gs-algo*) oder den Ordner (*languagetool-core.rules*). Gemessen wurden der Ausgangszustand, die Code-Zustände auf den fünf Branches nach den Refactorings und einer oder zwei Endzustände. Für den ersten Endzustand wurden vier der fünf Refactoring-Branches zusammengeführt: Design, Kontrollstrukturen, Datenstrukturen und der laut den Messungen energieeffizientere der beiden Energy-Smells-Branches.

Tabellen 4 und 5 zeigen die Mittelwerte der Messungen von Zeit und Energie je zehnmaliger Ausführung der Tests pro Code-Zustand. Je Spalte ist für jede Anwendung der niedrigste

Mittelwert grün unterlegt. Die Rohdaten, also die vollständigen Messergebnisse (je Durchlauf) aus den beiden Fallstudien, sind im Anhang unter A.2 zu finden.

	Zeit (Sek.)	Energie (Joule)
Ausgangszustand	93,5693752	1466,117816
Design	91,3801585	1429,488745
Kontrollstrukturen	92,5161549	1446,734125
Datenstrukturen	91,2445413	1411,155298
Energy Smells A	92,7490932	1446,511707
Energy Smells B	90,9707906	1416,465375
Endzustand ohne ESA	92,1818448	1430,636023
Endzustand mit ESA	93,134154	1444,721015

Tabelle 4: Durchschnittliche Messwerte von Zeit und Energie bei zehnmaliger Testausführung von *gs-algo*.

In Tabelle 4 ist zu sehen, dass bei GraphStream Energy Smells A mehr Energie benötigte (1446,51 Joule) als Energy Smells B (1416,47 Joule). Deshalb ergab sich der erste Endzustand ohne ESA (Energy Smells A). Da sich bei GraphStream bei Zusammenführung dieses Zustandes mit Energy Smells A keine Konflikte ergaben, wurde zusätzlich ein Endzustand mit ESA erzeugt und gemessen.

	Zeit (Sek.)	Energie (Joule)
Ausgangszustand	140,4361942	2179,697736
Design	140,8709791	2169,94649
Kontrollstrukturen	138,8594597	2154,453943
Datenstrukturen	139,123378	2157,602655
Energy Smells A	136,8033179	2122,66767
Energy Smells B	135,8006679	2123,75354
Endzustand ohne ESB	139,5110396	2171,465784

Tabelle 5: Durchschnittliche Messwerte von Zeit und Energie bei zehnmaliger Testausführung von *languagetool-core.rules*.

Bei LanguageTool benötigte Energy Smells A (2122,67 Joule) durchschnittlich knapp weniger Energie als Energy Smells B (2123,75 Joule), wie aus Tabelle 5 hervorgeht. Daher wurde hier der Endzustand ohne ESB (Energy Smells B) erzeugt. Da eine Zusammenführung mit Energy

Smells B zu Konflikten führte, wurde hier kein zweiter Endzustand erzeugt, der alle Kategorien beinhaltet hätte.

Aus Tabelle 4 geht hervor, dass sich die Durchschnittswerte der gemessenen Testdurchläufe je Code-Zustand bei GraphStream um bis zu rund 2,6 Sekunden und 55 Joule unterscheiden. Laut Tabelle 5 sind es bei LanguageTool rund 5 Sekunden und 58 Joule. Bei beiden Anwendungen weist der Code-Zustand Energy Smells B die geringste Laufzeit auf. Bei GraphStream benötigt der Zustand Datenstrukturen die geringste Energie, bei LanguageTool ist Energy Smells A am effizientesten.

5.1 Statistische Analyse der Ergebnisse

Im Folgenden werden die Messergebnisse hinsichtlich der zwei zentralen Forschungsfragen dieser Arbeit statistisch analysiert. Hierfür wurde das Tool OriginPro (https://www.originlab.com/) in der kostenlosen Studierendenversion verwendet. Anschließend werden die Resultate dieser Analyse ausgewertet. Die fachliche Interpretation erfolgt in Abschnitt 5.2.

Gibt es einen Unterschied zwischen den Code-Zuständen bezüglich der Energie oder Zeit?

Um eine Stichprobenmenge (wie hier die je zehn Messungen pro Code-Zustand) auf Unterschiede zu testen, wird in der statistischen Analyse für metrisch skalierte Werte eine einfaktorielle Varianzanalyse (ANOVA) durchgeführt. Sie kann hier dazu verwendet werden, um die Mittelwerte der Messergebnisse je Code-Zustand zu vergleichen und so zu ermitteln, ob sich mindestens einer von ihnen signifikant von den anderen unterscheidet. Als Referenz für die ANOVA, den darauffolgenden Tukey-Test und die Auswertung der beiden Verfahren dient Planing (2022).

Bei der Varianzanalyse werden zwei ungerichtete Hypothesen verwendet, nämlich H0 und H1, die im Kontext dieser Untersuchungen folgendermaßen lauten:

H0: Es gibt keinen Unterschied zwischen den einzelnen Code-Zuständen bezüglich des Energieverbrauchs beziehungsweise der Laufzeit.

H1: Es gibt mindestens einen Unterschied zwischen den einzelnen Code-Zuständen bezüglich des Energieverbrauchs beziehungsweise der Laufzeit.

Die Varianzanalyse gibt also Aufschluss darüber, ob mindestens ein Code-Zustand in Bezug auf Effizienz oder Performance von den übrigen signifikant abweicht. Wenn dies der Fall ist, wird ein sogenannter Post-hoc-Test durchgeführt, um diese Abweichungen zu identifizieren. Um zu erkennen, ob eine Gruppe (hier: ein Code-Zustand) signifikant von den anderen abweicht, wird bei der ANOVA sowohl der Mittelwert jeder einzelnen als auch der aller Gruppen berechnet. Der Mittelwert aller Gruppen wird auch als „Grand Mean“ bezeichnet. Anschließend wird für alle Strichprobenwerte (hier: die jeweils in einem Durchgang gemessenen Werte für Energie beziehungsweise Zeit) geprüft, ob der Mittelwert der eigenen Gruppe oder der Grand Mean den tatsächlichen Wert besser vorhersagt (sich also weniger von ihm unterscheidet). Wenn der Gruppenmittelwert den tatsächlichen Wert signifikant besser vorhersagt als der Grand Mean, weichen die Gruppenmittelwerte ebenfalls signifikant voneinander ab.

Voraussetzung für die Verwendung der ANOVA ist, dass die Daten innerhalb einer Gruppe metrisch skaliert und normalverteilt und die Gruppen untereinander unabhängig sind. Diese Voraussetzung sind hier erfüllt.

Ein Ergebnis der ANOVA ist der F-Wert. Er gibt in diesem Kontext Auskunft darüber, ob die Streuung der Werte auf die Refactoring-Maßnahmen oder den Zufall zurückzuführen sind. Hierzu wird das Verhältnis der Varianz zwischen den Gruppen zur Varianz innerhalb der Gruppen berechnet, die wiederum als mittlere Quadratsummen berechnet werden.

Wenn der F-Wert < 1, ist die Varianz innerhalb der Gruppen größer als die Varianz zwischen den Gruppen. Es liegt also eine breitere Streuung innerhalb der Gruppen vor als zwischen ihnen. Ist der F-Wert > 1, ist die Varianz zwischen den Gruppen größer als innerhalb. Ein großer F-Wert deutet also darauf hin, dass die Streuung der Werte durch tatsächliche Gruppenunterschiede bedingt ist.

Ein weiteres wichtiges Ergebnis der Varianzanalyse ist der *p*-Wert. Er steht für die Wahrscheinlichkeit, dass die tatsächlich gemessenen Ergebnisse auftreten, wenn H0 zutrifft (wenn es also keine Unterschiede zwischen den Code-Zuständen gibt). Das bedeutet, je niedriger der p-Wert, desto wahrscheinlich ist H1 (es bestehen Unterschiede zwischen den Gruppen). Es ist gängige

Praxis, für *p* ein Signifikanzniveau von 0,05 festzulegen, daher ist dies auch in dieser Arbeit der Fall.

Zusammenfassend bedeutet dies für die Varianzanalyse der gemessenen Energie- und Zeitwerte der beiden Anwendungen GraphStream und LanguageTool, dass ein hoher F-Wert > 1 und ein *p*-Wert ≤ 0,05 darauf schließen lassen, dass mindestens einer der Code-Zustände signifikant von den anderen abweicht. Eine Übersicht der *p*- und F-Werte ist in Tabelle 6 dargestellt. Die detaillierten Ergebnisse der deskriptiven Analyse samt ANOVA sind im Anhang unter A.3 und A.4 einzusehen.

	F-Wert	*p*
GraphStream		
Energie	7,99923	<0.0001
Zeit	6,9944	<0.0001
LanguageTool		
Energie	8,30535	<0.0001
Zeit	19,38436	<0.0001

Tabelle 6: *p*- und F-Werte der ANOVA für die beiden gemessenen Variablen Energie und Zeit für GraphStream und LanguageTool.

Anhand der Ergebnisse der ANOVA in Tabelle 6 wird deutlich, dass sich die durchschnittlichen Messwerte in den jeweiligen Zuständen sowohl bei GraphStream als auch bei LanguageTool bezüglich Zeit und Energie deutlich signifikant unterscheiden, da bei allen *p* weit unter dem Signifikanzniveau von 0,05 liegt und zudem ein hoher F-Wert ermittelt wurde. Bei LanguageTool liegt dieser für den Faktor Zeit sogar beinahe bei 20, was darauf hindeutet, dass mindestens einer der Code-Zustände eindeutig stark von den anderen bezüglich Performance abweichen muss.

Da die ANOVA nur über das Vorhandensein einer signifikanten Abweichung der Mittelwerte Aufschluss gibt, nicht aber, welche der Code-Zustände sich voneinander unterscheiden, wurde für die Faktoren Zeit und Energie bei beiden Anwendungen ein Post-hoc-Test durchgeführt, der die Mittelwerte der Code-Zustände paarweise miteinander vergleicht.

Bei dem hier gewählten Post-hoc-Test handelt es sich um den Tukey-Test, der häufig in Kombination mit einer ANOVA zum Einsatz kommt, da er robust und zuverlässig ist. Beim Tukey-

Test wird anhand der Differenzen zwischen den einzelnen Gruppenmittelwerten ein Intervall von Abweichungen ermittelt, die nicht als signifikant einzuschätzen sind. Zwei Code-Zustände werden nach dem Tukey-Test hinsichtlich des untersuchten Faktors (Energie oder Zeit) also als signifikant unterschiedlich eingestuft, wenn die Differenz der Mittelwerte dieser beiden Zustände außerhalb dieses Intervalls liegt. Signifikant unterschiedliche Gruppen wurden dann hier in einer Übersicht in sogenannten Grouping-Letters-Tables mit jeweils unterschiedlichen Buchstaben gekennzeichnet. Gruppen, die in derselben Buchstaben-Gruppe sind, weichen dem Tukey-Test nach also nicht signifikant voneinander ab (zu sehen in Tabellen 7 bis 10).

In ausführlicher Form sind die Ergebnisse des Tukey-Tests für GraphStream und Language-Tool jeweils für Energie und Zeit unter A.3 und A.4 zu finden.

	durchschnittl. Energie (Joule)	Gruppen	Gruppen	Gruppen
Ausgangszustand	1466,11782	A		
Kontrollstrukturen	1446,73412	A	B	
Energy Smells A	1446,51171	A	B	
Endzustand mit ESA	1444,72101	A	B	
Endzustand ohne ESA	1430,63602		B	C
Design	1429,48875		B	C
Energy Smells B	1416,46537			C
Datenstrukturen	1411,1553			C

Tabelle 7: Grouping-Letters-Table für die gemessenen Energie-Werte bei GraphStream.

Tabellen 7 und 8 zeigen die Grouping-Letters-Tables für GraphStream für die Faktoren Energie und Zeit. Hier ist zu sehen, dass der Ausgangszustand hinsichtlich beider gemessenen Variablen den höchsten und damit schlechtesten Mittelwert aufweist.

Laut Tabelle 7, die die Ergebnisse für den Energiebedarf abbildet, befinden sich die Code-Zustände Ausgangszustand, Kontrollstrukturen, Energy Smells A und dem Endzustand mit ESA in derselben Gruppe (A), und weichen somit nicht signifikant voneinander ab.

Die übrigen vier Zustände, nämlich der Endzustand ohne ESA, Design, Energy Smells B und Datenstrukturen, befinden sich hingegen allesamt in Gruppe C und weisen eine signifikante Abweichung von Gruppe A auf. Die Code-Zustände Energy Smells B und Datenstrukturen sind bei GraphStream eindeutig als am energieeffizientesten einzustufen, da sie sich ausschließlich in Gruppe C befinden.

	durchschnittl. Zeit (Sekunden)	Gruppen	Gruppen	Gruppen
Ausgangszustand	93,56938	A		
Endzustand mit ESA	93,13415	A		
Energy Smells A	92,74909	A	B	
Kontrollstrukturen	92,51615	A	B	C
Endzustand ohne ESA	92,18184	A	B	C
Design	91,38016		B	C
Datenstrukturen	91,24454		B	C
Energy Smells B	90,97079			C

Tabelle 8: Grouping-Letters-Table für die gemessenen Zeit-Werte bei GraphStream.

Bei der gemessenen Laufzeit für die unterschiedlichen Code-Zustände bei GraphStream bilden zusammen mit dem Ausgangszustand der Endzustand mit ESA, Energy Smells A, Kontrollstrukturen und der Endzustand ohne ESA eine Gruppe. Nur drei Zustände weichen also signifikant von dieser Gruppe bezüglich der Performance ab, da sie sich nicht in Gruppe A befinden: Design, Datenstrukturen und Energy Smells B. Als einziger Zustand, der sich ausschließlich in Gruppe C befindet, scheint Energy Smells B der vielversprechendste in Bezug auf Laufzeit zu sein.

Bei Betrachtung von Tabelle 7 und 8 fällt auf, dass die Endzustände mehr Zeit und Energie benötigen, obwohl sie die Änderungen der effizienteren Zustände Datenstrukturen, Energy Smells B und Design beinhalten (dies wird unter 5.2 diskutiert).

	durchschnittl. Energie (Joule)	Gruppen	Gruppen	Gruppen
Ausgangszustand	2179,69774	A		
Endzustand ohne ESB	2171,46578	A		
Design	2169,94649	A		
Datenstrukturen	2157,60266	A	B	
Kontrollstrukturen	2154,45394	A	B	C
Energy Smells B	2123,75354		B	C
Energy Smells A	2122,66767			C

Tabelle 9: Grouping-Letters-Table für die gemessenen Energie-Werte bei Language-Tool.

Auch bei LanguageTool benötigt der Ausgangszustand die meiste Energie, während ein Großteil der Code-Zustände nach den Refactorings diesbezüglich nicht signifikant von ihm abweichen. Tabelle 9 zeigt, dass alle bis auf die beiden Energy-Smells-Zustände sich in Gruppe A befinden. Energy Smells A ist der Einzige, der sich hier nur in Gruppe C befindet und ist damit laut den Durchschnittswerten auch am energiesparendsten.

	durchschnittl. Zeit (Sekunden)	Gruppen	Gruppen	Gruppen
Design	140,87098	A		
Ausgangszustand	140,43619	A	B	
Endzustand ohne ESB	139,51104	A	B	
Datenstrukturen	139,12338	A	B	
Kontrollstrukturen	138,85946		B	
Energy Smells A	136,80332			C
Energy Smells B	135,80067			C

Tabelle 10: Grouping-Letters-Table für die gemessenen Zeit-Werte bei LanguageTool.

In Tabelle 10 ist zu sehen, dass nur bei der durchschnittlichen gemessenen Zeit für LanguageTool der Zustand Design einen noch höheren (und damit schlechteren) Wert aufweist als der Ausgangszustand. Die Gruppe A, innerhalb derer es keine signifikanten Unterschiede zwischen den Mittelwerten für die Laufzeit gibt, umfasst neben diesen beiden Code-Zuständen noch den Endzustand ohne ESB und Datenstrukturen. Kontrollstrukturen befindet sich nur in der Gruppe B, in der sich allerdings auch der Ausgangszustand befindet. Kontrollstrukturen weicht also bezüglich der Performance ebenfalls nicht signifikant vom Ausgangszustand ab.

Einzig Energy Smells A und B bilden die Gruppe C, unterscheiden sich also nicht signifikant voneinander, sind aber beide im Durchschnitt schneller als die übrigen Code-Zustände.

Die Interpretation und Besprechung möglicher Gründe für Besonderheiten, die aus den Tabellen 7 bis 10 und ihrer Auswertung hervorgegangen sind, erfolgen in Abschnitt 5.2.

Besteht eine Korrelation zwischen Zeit und Energie?

Der Korrelationskoeffizient *r* nach Pearson gibt Auskunft darüber, ob ein Zusammenhang zwischen zwei metrisch skalierten Variablen besteht; in diesem Falle zwischen dem Energie- und

dem Zeitbedarf für je einen Messdurchlauf. Wie bei der ANOVA dient auch hier Planing (2022) als Referenz.

Der Wert von *r* liegt immer zwischen -1 und 1, wobei -1 und 1 einen vollständigen linearen Zusammenhang (negative oder positive Korrelation) bedeuten, während bei 0 überhaupt kein Zusammenhang zwischen den Variablen besteht. Auch hier wird der *p*-Wert angegeben, anhand dessen bestimmt werden kann, ob der Zusammenhang als signifikant eingestuft wird. Tabelle 11 zeigt die Ergebnisse des Pearson-Tests für beide Anwendungen.

	r	*p*
GraphStream	0,84227	< 0,0001
LanguageTool	0,69663	< 0,0001

Tabelle 11: Korrelationskoeffizient und *p*-Wert nach Pearson für GraphStream und LanguageTool bezüglich des Zusammenhangs zwischen Energie und Zeit.

Hier ist zu sehen, dass in beiden Fallstudien ein positiver Zusammenhang zwischen Zeit und Energie besteht, wobei dieser bei GraphStream stärker ausgeprägt ist als bei LanguageTool. Die Signifikanz dieses Zusammenhangs wird durch die *p*-Werte, die kleiner als 0,05 sind, bestätigt.

5.2 Diskussion

An dieser Stelle werden die Ergebnisse aus der statistischen Analyse im vorangegangenen Abschnitt fachlich interpretiert.

5.2.1 Effektivität der Refactoring-Kategorien

Dieser Abschnitt konzentriert sich auf die Energieeffizienz der fünf Refactoring-Branches, auf denen sich die zu den jeweiligen Kategorien gehörigen Code-Zustände befinden. Die beiden Anwendungen GraphStream und LanguageTool werden hinsichtlich dieses Aspekts

weitgehend unabhängig voneinander betrachtet, da sie aufgrund des unterschiedlichen Ausmaßes der einzelnen Refactorings nicht sinnvoll miteinander verglichen werden können. (Mehr dazu unter 5.3.)

GraphStream

Die statistische Analyse und ihre Auswertung im vorherigen Abschnitt haben gezeigt, dass nicht alle Code-Zustände signifikant bessere Ergebnisse erzielen konnten als der Ausgangszustand. Bei GraphStream waren bezüglich des Energieverbrauchs der Endzustand ohne ESA und Design und in noch deutlicherem Maße Energy Smells B und Datenstrukturen besser als der Ausgangszustand. Eine signifikant geringere Laufzeit als der Ausgangszustand hatten die Zustände Design, Datenstrukturen und Energy Smells B, wobei sich auch hier Energy Smells B von den anderen abhob.

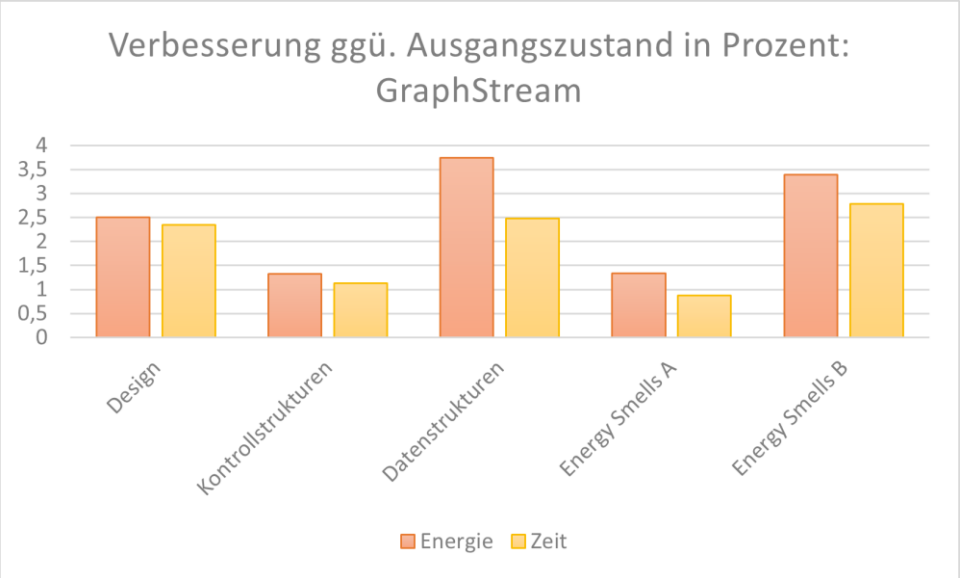


Abbildung 6: Steigerung der Energie- und Zeiteffizienz der fünf Kategorien-Branches gegenüber dem Ausgangszustand anhand der Mittelwerte (GraphStream).

Für eine bessere Einordnung und Übersicht sind in Abbildung 6 für die fünf Refactoring-Techniken die jeweiligen prozentualen Abweichungen zum Ausgangszustand in einem Balkendiagramm abgebildet (analog für LanguageTool in Abibldung 7). Basis von Abbildung 6 und 7

bilden die Mittelwerte, die auch in den Grouping-Letters-Tables (Tabellen 7 bis 10) aufgeführt sind.

Durch Abbildung 6 wird bestätigt, dass die Kategorien Datenstrukturen und Energy Smells B bei GraphStream den größten Effekt bezüglich der Reduktion des Energieverbrauchs zu haben scheinen. Ihre Verbesserung gegenüber dem Ausgangszustand liegt über 3%.

Tatsächlich war das Ausmaß der Code-Änderungen durch den Austausch von Datenstrukturen vergleichsweise hoch. (Wie in 4.3.1 beschrieben, wurden 30 Dateien modifiziert.) Es erscheint also nachvollziehbar, dass in einer Anwendung wie GraphStream, in der viele Collection-Datenstrukturen Verwendung finden, eine auf Effizienz abzielende Wahl dieser Strukturen, wie sie unter RNr. 9 und 10 beschrieben ist, den Energieverbrauch positiv beeinflussen kann.

Auffällig in Abbildung 6 ist der starke Unterschied zwischen den Kategorien Energy Smells A und B. Zur Erinnerung: Die beiden Kategorien umfassen die Techniken RNr. 12, 14, 15, 16, 18 und 19. Energy Smells A enthält zusätzlich RNr. 13 (Umwandlung nicht-statischer Methoden in statische). Energy Smells B enthält zusätzlich RNr. 17 (Umwandlung statischer Variablen in nicht-statische).

Leider besteht kein messbarer Code-Zustand, in dem keines dieser beiden Refactorings neben den anderen Energy Smells umgesetzt wurde, und der Aufwand, einen solchen zu erzeugen, ist im Rahmen der vorliegenden Arbeit nicht mehr möglich. Daher kann an dieser Stelle nicht erschlossen werden, in welchem Maße die Refactorings nach RNr. 13 und RNr. 17 möglicherweise zu einem höheren beziehungsweise niedrigeren Energieverbrauch beigetragen haben. Fest steht nur, dass die Kategorie Energy Smells B, bei der statische Variablen in nicht-statische umgewandelt wurden, laut den Messergebnissen wesentlich weniger Energie verbrauchte als die drei Branches Design, Kontrollstrukturen und Energy Smells A. Da diese Kategorie Refactorings unterschiedlicher Art zusammenfasst, lassen sich keine Aussagen über die Wirkung einzelner Techniken treffen. In Summe scheinen sie jedoch effektiv zu sein.

Mit 2,5% Effizienzsteigerung und sich auch nach dem Tukey-Test in der effizientesten Gruppe befindend (siehe Tabelle 7) weist auch das Design-Refactoring gute Ergebnisse hinsichtlich der Energieeffizienz auf. Da hier hauptsächlich Getter und Setter entfernt wurden, scheinen hier die Ergebnisse von Georgiou et al. (2020) und Tonini et al. (2013) bestätigt zu werden.

Die verbleibenden Kategorien Energy Smells A und Kontrollstrukturen benötigen laut den Messergebnissen zwar weniger Energie als der Ausgangszustand, nicht aber in signifikantem Maße. Energy Smells A wurde oben bereits besprochen. Der geringe Effekt durch die Code-Änderungen bezüglich der Kontrollstrukturen kann bedeuten, dass die Refactoring-Techniken dieser Kategorie zu vernachlässigen sind, da es sich bei *gs-algo* um ein Projekt handelt, in dem Kontrollstrukturen durchaus eine große Rolle spielen.

LanguageTool

Bei LanguageTool konnten die beiden Kategorien Energy Smells A und B die höchste Energieeffizienzsteigerung erzielen. Auch hier lässt sich allerdings nicht sagen, ob RNr. 13 und RNr. 17 sich gegenseitig ausgleichen oder vielleicht sogar gar keinen Effekt haben. Abgesehen von diesen beiden Techniken wurden in der Kategorie Energy Smells vor allem mehrfache Array-Zugriffe auf denselben Index durch einmaligen Zugriff und Speicherung in einer Variable (RNr. 15) und String-Konkatenation mit StringBuilder durch Verwendung des +-Operators ersetzt (RNr. 19). Es ist also wahrscheinlich, dass mindestens eines dieser beiden Refactorings einen starken Einfluss auf den Energiebedarf hat.

Den Messungen zufolge waren die Änderungen bezüglich Kontrollstrukturen und Datenstrukturen deutlich weniger effektiv, obwohl hier verhältnismäßig viele Dateien modifiziert wurden. Da die Refactoring-Techniken aus diesen beiden Kategorien häufig im Kontext von Schleifen und Iterationen Anwendung finden, ist es möglich, dass hier der Effekt größer gewesen wäre, wenn die betreffenden Instanzen der Collection-Implementierungen in den Tests mit mehr Objekten befüllt und so die betreffenden Stellen häufiger ausgeführt worden wären.

Da die Kategorie Kontrollstrukturen in keiner der beiden Anwendungen eine signifikante Steigerung der Energieeffizienz gegenüber dem Ausgangszustand erzielen konnte, steht jedoch auch die Vermutung im Raum, dass der Effekt der in dieser Kategorie angewendeten Refactoring-Techniken insgesamt als eher gering einzustufen ist.

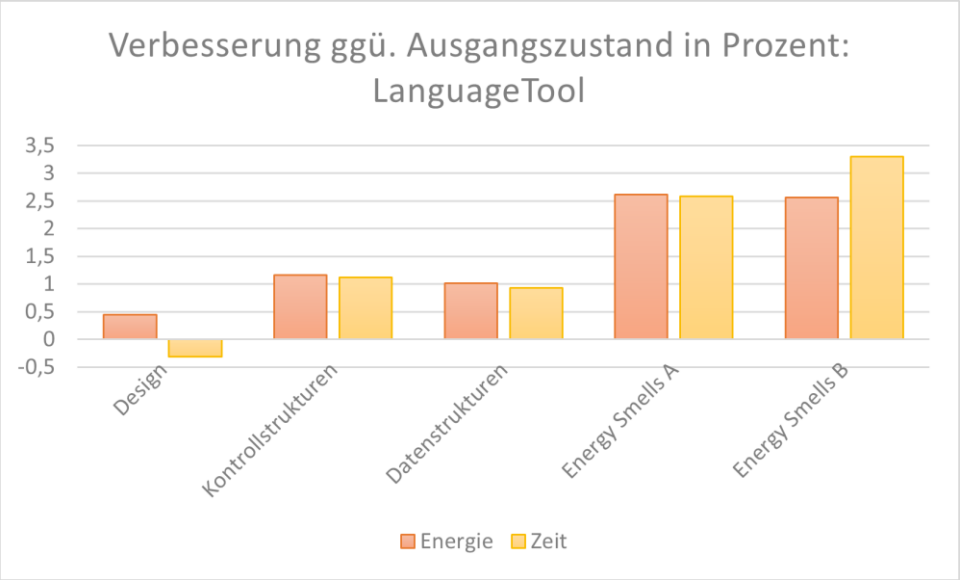


Abbildung 7: Steigerung der Energie- und Zeiteffizienz der fünf Kategorien-Branches gegenüber dem Ausgangszustand anhand der Mittelwerte (LanguageTool).

Der Design-Branch benötigt von allen Refactoring-Branches die meiste Energie. Tatsächlich unterscheiden sich sein gemessener Energiebedarf, ebenso wie der von den Branches Kontrollstrukturen und Datenstrukturen, nicht signifikant von dem des Ausgangszustands. Das Entfernen von Gettern und Settern, welches auch hier eine große Anzahl an Dateien betraf, hat bei LanguageTool anscheinend kaum Wirkung gezeigt.

5.2.2 Wechselwirkung zwischen den Refactorings

Bei Betrachtung der Tabellen 7 bis 10 fällt auf, dass bei beiden Anwendungen zwar alle Refactoring-Branches einen geringeren Energieverbrauch aufweisen als der Ausgangszustand, jedoch die Zusammenführung dieser Branches nicht die höchste Energieeffizienz zur Folge hat. Im Gegenteil benötigt der Endzustand bei LanguageTool an zweiter Stelle nach dem Ausgangszustand sogar die meiste Energie und unterscheidet sich mit seinen Messwerten nicht signifikant von ihm. Bei GraphStream befinden sich die Endzustände nur im mittleren Bereich. Nur der Endzustand ohne Energy Smells A unterscheidet sich im Energieverbrauch signifikant vom Ausgangszustand.

Die Effekte, die den Energiebedarf reduzieren, summieren sich den Messungen zufolge also nicht auf, sondern scheinen sich sogar gegenseitig zu behindern. Zur Untersuchung dieses Umstandes wären Messwerte für den Energieverbrauch der einzelnen Kategorien jeweils in Kombination miteinander vonnöten, die aber (abgesehen von Energy Smells bei GraphStream) leider nicht Teil der Experimente dieser Arbeit waren. Allerdings eröffnet sich mit diesen Ergebnissen eine spannende Richtung für zukünftige Forschung.

5.2.3 Zusammenhang zwischen Laufzeit und Energieeffizienz

Der Korrelationskoeffizient für Zeit und Energie beträgt nach Tabelle 11 für GraphStream 0,84227 und für LanguageTool 0,69663. Beide Werte weisen auf eine positive Korrelation hin, wobei dieser Zusammenhang bei LanguageTool etwas weniger stark ausgeprägt zu sein scheint. Es lässt sich folgern, dass den Messergebnissen nach ein hoher Energieverbrauch tendenziell mit einer hohen Laufzeit einhergeht und umgekehrt.

Die Refactoring-Techniken, die im Rahmen dieser Arbeit angewendet wurden, zielen zunächst nur auf eine Steigerung der Energieeffizienz ab. In Abbildung 6 und 7 ist zu sehen, dass dies in den meisten Fällen eine Steigerung der Performance zur Folge hatte.

Bei LanguageTool (Abbildung 7) allerdings fallen diesbezüglich zwei Besonderheiten auf: Von den Refactoring-Branches beider Anwendungen ist Energy Smells B der einzige, bei dem eine höhere Steigerung in der Performance als in der Energieeffizienz ermittelt wurde. Sein Energieverbrauch unterscheidet sich von dem von Energy Smells A nicht nennenswert, die Performance ist aber deutlich höher. Da zudem der einzige Unterschied zwischen diesen beiden Branches die Anwendung von RNr. 13 beziehungsweise RNr. 17 ist, ist zu vermuten, dass eine dieser beiden Techniken einen großen Einfluss auf die Laufzeit, also auf die Performance, hat. Ähnlich wie in 5.2.1 besprochen kann RNr. 13 (Umwandlung nicht-statischer Methoden in statische) einen negativen oder RNr. 17 (Umwandlung statischer Variablen in nicht-statische) einen positiven Einfluss auf die Laufzeit haben.

Die andere Besonderheit bei LanguageTool ist die gemessene Verschlechterung der Performance in der Kategorie Design. Dieser Code-Zustand ist der Einzige, der langsamer ist als der

Ausgangszustand (wenn auch nicht in signifikantem Maße). Da hier lediglich genau wie bei GraphStream Getter und Setter entfernt und ihre Verwendung durch direkten Zugriff ersetzt wurde, bei GraphStream jedoch deutliche Verbesserungen in Bezug auf Energieeffizienz und Performance aus den Messergebnissen abzulesen sind, lässt sich dieses Phänomen ohne weitere Untersuchungen nicht erklären. Selbstverständlich kann auch die Möglichkeit von Messfehlern nicht gänzlich ausgeschlossen werden.

Trotz dieser beiden Ausnahmen lässt sich an dieser Stelle festhalten, dass es wahrscheinlich ist, dass ein Refactoring für Performance auch ein Refactoring für Energieeffizienz ist. Bewiesen ist dieses Verhältnis aber nicht, da hier nur Code-Änderungen mit dem Ziel des geringeren Energiebedarfs durchgeführt wurden, woraus eine kürzere Laufzeit folgte. Die gegenteilige Richtung wurde nicht überprüft.

5.3 Kritische Betrachtungen

Die praktische Durchführung der Fallstudien unterlag einigen Einschränkungen und Schwierigkeiten, die es in zukünftigen Arbeiten dieser Art so weit wie möglich zu beheben gilt. Diese werden im Folgenden besprochen.

5.3.1 Vergleichbarkeit

Einer dieser Faktoren ist die mangelnde Vergleichbarkeit der Ergebnisse auf mehreren Ebenen. Zum einen können, wie zu Anfang in 5.2.1 erwähnt, die Messergebnisse der Anwendungen untereinander nicht hinsichtlich ihres Effekts verglichen werden, da der Refactoring-Bedarf je Technik für jede Anwendung unterschiedlich ist. Wenn beispielsweise in einer Anwendung für eine Kategorie nur eine einzige Zeile geändert werden muss, während für dieselbe Kategorie in der anderen Anwendung über hundert Zeilen angepasst werden, ist ein Vergleich hinsichtlich des Effekts hinfällig. Aus demselben Grund ist auch ein Vergleich des Energiebedarfs unterschiedlicher Refactoring-Branches in derselben Anwendung nur bedingt aussagekräftig, da auch die Kategorien nicht gleich viele Änderungen erfordern. Wenn ein Branch

energieeffizienter ist als ein anderer, lässt dies nur Schlüsse zu, die sich konkret auf die jeweilige Anwendung beziehen. Um allgemeingültige Schlüsse bezüglich der Wirksamkeit der Refactorings ziehen zu können, müssten sie bei einer wesentlich größeren Anzahl von Anwendungen umgesetzt und gemessen werden.

Zudem werden in den Tests und innerhalb des produktiven Codes manche der modifizierten Code-Stellen häufiger aufgerufen als andere. Auch dies kann den Eindruck der Auswirkungen verzerren.

5.3.2 Effekte einzelner Techniken

Die in dieser Arbeit angewendete Methodik berücksichtigt nicht den Effekt, den einzelne Refactoring-Techniken haben, sondern betrachtet mit Ausnahme der Energy-Smells-Varianten nur die Auswirkungen der Kategorien, die immer mehrere Techniken umfassen. Die Messergebnisse legen nahe, dass Refactoring-Techniken sich möglicherweise gegenseitig behindern können, sodass sie, wenn sie gleichzeitig angewendet werden, ein weniger gutes Ergebnis in Bezug auf die Energieeffizienz erzielen, als wenn nur eine dieser Techniken verwendet worden wäre. Solche Effekte können mit dem methodischen Aufbau in dieser Arbeit nicht identifiziert werden.

5.3.3 Zuverlässigkeit der Messungen

Die Messung des Energieverbrauchs lokaler Software-Anwendungen ist ein komplexes Thema, das noch Gegenstand aktueller Forschung ist (siehe 2.4.2 und 4.2). Die Messung auf einem Windows-Laptop erwies sich trotz aller Maßnahmen als fehleranfällig, da das System beispielsweise mitten in den Messungen Hintergrundprozesse startete. Eine andere Form der Energieverbrauchsmessung, beispielweise auf einer separaten Maschine mit möglichst wenig Hintergrundprozessen, die optimalerweise auch die Erhebung eines größeren Datensatzes ermöglicht, könnte zuverlässigere Ergebnisse liefern.

6 Fazit und Ausblick

Im Fokus dieser Arbeit standen zwei Fragen:

1) Wie stark kann durch Refactorings die Energieeffizienz von Java-Anwendungen verbessert werden?
2) Wie sieht der Zusammenhang zwischen Laufzeit und Energieeffizienz aus?

Um diese Fragen zu beantworten, wurden in einer umfangreichen Literaturrecherche 19 Refactoring-Techniken ermittelt, die auf eine Senkung des Energiebedarfs von Java-Anwendungen abzielen. Diese 19 Techniken wurden in fünf Kategorien eingeteilt: Design, Kontrollstrukturen, Datenstrukturen und Energy Smells A und B.

Die zwei open-source Java-Anwendungen GraphStream und LanguageTool wurden ausgewählt, um an ihnen die Refactorings durchzuführen und jeweils den Energieverbrauch und die benötigte Zeit der unterschiedlichen Code-Zustände für zehnmaliges Ausführen ihrer Unit-Tests zu messen. Die Ergebnisse dieser Messungen wurden hinsichtlich der statistischen Signifikanz in den Unterschieden zwischen den Code-Zuständen und auch der Korrelation zwischen Zeit- und Energieaufwand untersucht.

Die Ergebnisse dieser Untersuchungen legen nahe, dass das Ausmaß, in welchem die Energieeffizienz einer Anwendung durch Refactoring-Techniken gesteigert werden kann, von Anwendung zu Anwendung variiert. Bei GraphStream erzielten diesbezüglich die Refactoring-Kategorien Design, Energy Smells B und Datenstrukturen signifikante Verbesserungen, bei LanguageTool waren es Energy Smells A und Energy Smells B.

Es gibt wahrscheinlich nicht die eine Kategorie von Energie-Refactorings, die bei allen Anwendungen zu signifikanten Einsparungen führt. Wenn man dennoch eine nennen wollte, wäre die vielversprechendste Kategorie diesbezüglich Energy Smells B, da sie als einzige bei beiden Anwendungen eine signifikante Steigerung der Energieeffizienz bewirkte. Um diese These zu

überprüfen, wird weitere Forschung vonnöten sein, die eine größere Anzahl von Fallstudien umfasst.

Außerdem muss auch die Übertragbarkeit und Skalierbarkeit zu berücksichtigt werden. Die Beurteilung des Energiebedarfs während der Ausführung von Unit-Tests bietet nur begrenzte Einblicke in die tatsächlichen Auswirkungen der Refactoring-Maßnahmen auf potenzielle reale Szenarien. Eine realistische Bewertung des Effekts könnte durch Informationen darüber erleichtert werden, welche Teile einer Anwendung von wie vielen Nutzern und mit welcher Häufigkeit ausgeführt werden. Weiterhin ist es wichtig zu beachten, dass weit verbreitete Anwendungen, die in größerem Umfang verwendet werden, das Potenzial haben, in der Gesamtbilanz eine größere Energieeinsparung zu bewirken.

Eine unerwartete Schlussfolgerung aus den Resultaten der Fallstudien ist die Tatsache, dass die Kombination von Refactoring-Kategorien, die einzeln jeweils zu einem geringeren Energieverbrauch führten, in Summe zu einem ähnlich ineffizienten Zustand führten wie der Ausgangszustand, der keinem Refactoring unterzogen wurde. Hier sind eingehendere Untersuchungen, die die Auswirkungen unterschiedlicher Kombinationen von Refactoring-Techniken betrachten, für zukünftige Forschung vorstellbar.

Zwischen Laufzeit und Energieverbrauch scheint ein positiver Zusammenhang zu bestehen. Eine Verbesserung der Energieeffizienz auf den einzelnen Refactoring-Branches hatte in fast allen Fällen eine Steigerung der Performance zur Folge. Zwar scheint es naheliegend, dass andersherum auch ein Refactoring für Performance einen geringeren Energiebedarf bewirkt, allerdings ist dies kaum untersucht, da sich die Studien zu diesem Thema häufig entweder auf den Energieverbrauch oder die Performance konzentrieren und selten beides auswerten.

Zur Untersuchung, ob bei einem Refactoring für mehr Performance auch eine höhere Energieeffizienz zur Folge hat, wäre ein ähnlicher methodischer Aufbau wie in dieser Bachelorarbeit möglich; mit dem Unterschied, dass nur Refactoring-Techniken verwendet werden, die nachgewiesenermaßen zu einer geringeren Laufzeit beitragen. Die in Abschnitt 2.4.2 diskutierten Profiler wären an dieser Stelle ebenfalls ein geeignetes Werkzeug. Anschließend könnte auch dort die Korrelation zwischen Zeit- und Energieaufwand ermittelt werden. Wenn sich der Zusammenhang als ähnlich hoch erweisen würde wie in der vorliegenden Arbeit, könnte man Refactorings für Performance und Energieeffizienz gesammelt betrachten.

Softwareentwickler:innen, die eine möglichst ressourcenschonende Anwendung zum Ziel haben, könnten sich die bereits vorhandenen Profiler zunutze machen, in dem Wissen, dass ihre Anwendung weniger Energie benötigen wird, auch wenn sie Programmier- und Refactoring-Techniken verwenden, die ursprünglich nur auf Performance abzielen.

Die Motivation für die Formulierung der beiden zentralen Fragen dieser Bachelorarbeit resultierte aus ihrer Relevanz für die nachhaltige Softwareentwicklung. Bewusst wurde Java als Programmiersprache gewählt, da sie nach wie vor breit vertreten ist und effizienzsteigernde Refactoring-Techniken daher in zahlreichen Programmen eingesetzt werden könnten. Voraussetzung hierfür ist, dass sich dieser Aufwand lohnt. Ist das Energiesparpotenzial zu gering, sind Zeit und Mühe der Entwickler:innen womöglich in einer der anderen Stationen des Software-Lebenszyklus‘ sinnvoller einzusetzen. Wie Georgiou et al. (2020) in ihrer Übersichtsarbeit aufgezeigt haben, gibt es zahlreiche Aspekte, in denen die Nachhaltigkeit von Software verbessert werden kann. In dieser Arbeit wurde nur einer von vielen untersucht.

Wie groß der Effekt von Refactorings für mehr Energieeffizienz sein kann, lässt sich anhand der zwei Fallstudien nicht abschließend sagen. Da dieses Forschungsgebiet von anhaltendem wissenschaftlichem Interesse ist, bietet diese Arbeit jedoch eine vielversprechende und relevante Grundlage für weitere Untersuchungen.

Quellenverzeichnis

Agarwal, S., Nath, A. & Chowdhury, D. (2012) „Sustainable Approaches and Good Practices in Green Software Engineering“, *International Journal of Research and Reviews in Computer Science (IJRRCS)*, Vol. 3, No. 1.

Bree, D. C. & Cinneide, M. O. (2022) „Removing Decorator to Improve Energy Efficiency“, *2022 IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER).* Honolulu, HI, USA, 15.03.2022 - 18.03.2022, IEEE, S. 902–912.

Bree, D. C. & Cinnéide, M. Ó. (2020) „Inheritance versus Delegation“, *Proceedings of the IEEE/ACM 42nd International Conference on Software Engineering Workshops.* Seoul Republic of Korea, 27 06 2020 19 07 2020. New York, NY, USA, ACM, S. 323–329.

Bree, D. C. & Cinnéide, M. Ó. (2021) „Automated Refactoring for Energy-Aware Software“, *2021 IEEE International Conference on Software Maintenance and Evolution (ICSME).* Luxembourg, 27.09.2021 - 01.10.2021, IEEE, S. 689–694.

Cruz, L. & Abreu, R. (2019) „Improving Energy Efficiency Through Automatic Refactoring“, *Journal of Software Engineering Research and Development*, Vol. 7, S. 2.

Fowler, M. (2020) *Refactoring -- Wie Sie das Design bestehender Software verbessern*, Mitp Verlag.

Georgiou, S., Rizou, S. & Spinellis, D. (2020) „Software Development Lifecycle for Energy Efficiency“, *ACM Computing Surveys*, Vol. 52, No. 4, S. 1–33.

Gupta, U, Kim, YG, Lee, S, Tse, J, Lee, H-HS, Wei, G-Y, Brooks, D & Wu, C-J (Hg.) (2021) *2021 IEEE International Symposium on High-Performance Computer Architecture (HPCA)*, IEEE.

Hasan, S., King, Z., Hafiz, M., Sayagh, M., Adams, B. & Hindle, A. (2016) „Energy profiles of Java collections classes“, *Proceedings of the 38th International Conference on Software Engineering*. Austin Texas, 14 05 2016 22 05 2016. New York, NY, USA, ACM, S. 225–236.

Hassan, H. H. M., Shawky, A. & Farag, I. (2017) „Performance vs. Power and Energy Consumption: Impact of Coding Style and Compiler“, *International Journal of Advanced Computer Science and Applications*, Vol. 8, No. 12.

Imran, A., Kosar, T., Zola, J. & Bulut, M. F. (2023) *Predicting the Impact of Batch Refactoring Code Smells on Application Resource Consumption*.

Kim, D., Hong, J.-E., Yoon, I. & Lee, S.-H. (2018) „Code refactoring techniques for reducing energy consumption in embedded computing environment“, *Cluster Computing*, Vol. 21, No. 1, S. 1079–1095.

Kruglov, A., Succi, G. & Dlamini, G. (2023) „System Energy Consumption Measurement“, in Kruglov, A. & Succi, G. (Hg.) *Developing Sustainable and Energy-Efficient Software Systems*, Cham, Springer International Publishing, S. 27–38.

Kumar, M., Li, Y. & Shi, W. (2017) „Energy consumption in Java: An early experience“, *2017 Eighth International Green and Sustainable Computing Conference (IGSC)*. Orlando, FL, 23.10.2017 - 25.10.2017, IEEE, S. 1–8.

Lima, L. G., Soares-Neto, F., Lieuthier, P., Castor, F., Melfe, G. & Fernandes, J. P. (2016) „Haskell in Green Land: Analyzing the Energy Behavior of a Purely Functional Language“, *2016 IEEE 23rd International Conference on Software Analysis, Evolution, and Reengineering (SANER)*. Suita, 14.03.2016 - 18.03.2016, IEEE, S. 517–528.

Long, H. D., Vergilio, T. & Kor, A.-L. (2023) *Comparative Performance and Energy Efficiency Analysis of Jvm Variants and Graalvm in Java Applications*.

Murugesan, S. (2008) „Harnessing Green IT: Principles and Practices“, *IT Professional*, Vol. 10, No. 1, S. 24–33.

Palomba, F., Di Nucci, D., Panichella, A., Zaidman, A. & Lucia, A. de (2019) „On the impact of code smells on the energy consumption of mobile applications“, *Information and Software Technology*, Vol. 105, S. 43–55.

Pereira, R., Carção, T., Couto, M., Cunha, J., Fernandes, J. P. & Saraiva, J. (2020) „SPELLing out energy leaks: Aiding developers locate energy inefficient code“, *Journal of Systems and Software*, Vol. 161, S. 110463 [Online]. DOI: 10.1016/j.jss.2019.110463.

Pereira, R., Couto, M., Ribeiro, F., Rua, R., Cunha, J., Fernandes, J. P. & Saraiva, J. (2017) „Energy efficiency across programming languages: how do energy, time, and memory relate?“, *Proceedings of the 10th ACM SIGPLAN International Conference on Software Language Engineering.* Vancouver BC Canada, 23 10 2017 24 10 2017. New York, NY, USA, ACM, S. 256–267.

Pereira, R., Couto, M., Ribeiro, F., Rua, R., Cunha, J., Fernandes, J. P. & Saraiva, J. (2021) „Ranking programming languages by energy efficiency“, *Science of Computer Programming*, Vol. 205, S. 102609.

Pereira, R., Couto, M., Saraiva, J., Cunha, J. & Fernandes, J. P. (2016) „The influence of the Java collection framework on overall energy consumption“, *Proceedings of the 5th International Workshop on Green and Sustainable Software.* Austin Texas, 14 05 2016 22 05 2016. New York, NY, USA, ACM, S. 15–21.

Petri, B. & Petri, B. (2023) *Java-Grundlagen* [Online]. Verfügbar unter https://www.java-tutorial.org/java-grundlagen.html (Abgerufen am 29 Juli 2023).

Pinto, G., Castor, F. & Liu, Y. D. (2014) „Understanding energy behaviors of thread management constructs“, *Proceedings of the 2014 ACM International Conference on Object Oriented Programming Systems Languages & Applications.* Portland Oregon USA, 20 10 2014 24 10 2014. New York, NY, USA, ACM, S. 345–360.

Planing, P. (2022) *Statistik Grundlagen: Das neue Lehrbuch mit über 200 YouTube-Videos rund um die Burgerkette FIVE PROFS*, Stuttgart, Planing Publishing.

Programiz (2023) *Java JDK, JRE and JVM* [Online]. Verfügbar unter https://www.programiz.com/java-programming/jvm-jre-jdk (Abgerufen am 28 Juli 2023).

Şanlıalp, İ., Öztürk, M. M. & Yiğit, T. (2022) „Energy Efficiency Analysis of Code Refactoring Techniques for Green and Sustainable Software in Portable Devices“, *Electronics*, Vol. 11, No. 3, S. 442.

Tonini, A. R., Fischer, L. M., Mattos, J. C. B. de & Brisolara, L. B. de (2013) „Analysis and Evaluation of the Android Best Practices Impact on the Efficiency of Mobile Applications“, *2013 III Brazilian Symposium on Computing Systems Engineering.* Niteroi, Rio De Janeiro, Brazil, 04.12.2013 - 08.12.2013, IEEE, S. 157–158.

Trefethen, A. E. & Thiyagalingam, J. (2013) „Energy-aware software: Challenges, opportunities and strategies“, *Journal of Computational Science*, Vol. 4, No. 6, S. 444–449.

Yuki, T. & Rajopadhye, S. (2014) „Folklore Confirmed: Compiling for Speed = Compiling for Energy“, in Caşcaval, C. & Montesinos, P. (Hg.) *Languages and Compilers for Parallel Computing*, Cham, Springer International Publishing, S. 169–184.

A Anhang

A.1 Refactoring: Reguläre Ausdrücke

Regel-Nr.	Technik	Regex
	Design	
RNr. 1	Reduzierung von Objekten (bei Observer, Abstract Factory und Decorator)	*^(?!(\s**)|import|package).*(observer|observable|notify|subscribe|abstract|factory|creator|generator|decorator)*
RNr. 2	Reduzierung von Funktionsaufrufen (bei Observer, Abstract Factory und Decorator)	
RNr. 3	Ersetzen von Delegation durch Vererbung	*public\s[^\s]*\b\(.*\)\s?\{\n.*new*
RNr. 4	Entfernen von Gettern und Settern	*public.+(?!tar)get.+\{\n\s+return\s(?!.*\.)* *public void.+set.+\{\n\s*(?!.+\()*
	Kontrollstrukturen	
RNr. 5	Verwendung von for-each-Konstrukt bei allen Collections außer ArrayList	*^(?!(\s**)).*\bfor \(int*
RNr. 6	Verwendung von for-Schleife mit Länge in externer Variable bei ArrayList	
RNr. 7	Vereinfachung verschachtelter Schleifen	*for.+\{\n\s*for*
RNr. 8	Anpassung der Reihenfolge innerhalb von Bedingungen	*((||)|(&&))*
	Datenstrukturen	
RNr. 9	Kontextabhängiger Austausch von Collection-Datenstrukturen	*Listen: new.+List<* *Maps: new.+Map< (Ausnahme: Hashtable statt HashMap)* *Sets: ^(?!.*THashSet<).*Set<*
RNr. 10	Austausch von Datenstrukturimplementierungen	
RNr. 11	Ersetzen nebenläufiger Datenstrukturen durch nicht-nebenläufige	*ConcurrentSkipListSet|CopyOnWriteArrayList|ConcurrentHashMap|ConcurrentSkipListMap|CopyOnWriteArraySet*

	(Weitere) Energy Smells	
RNr. 12	Überprüfen (und ggf. Beenden) von Threads, die bisher nicht explizit beendet werden	*new Thread*
RNr. 13	Umwandlung nicht-statischer Methoden in statische, wenn sie auf keine Instanzvariablen ihrer Klasse zugreifen	(Wird mit IntelliJ-Tool 'Code Inspection' ermittelt. Java > Perfomance > Method may be static
RNr. 14	Entfernen von ‚totem' (ungenutztem) Code	(Wird mit IntelliJ-Tool 'Code Inspection' ermittelt, nachdem bereits alle Tests in die Refactoring-Branches gemerget wurden, da das Entfernen von ungenutztem Code keine weiteren Änderungen oder Ergänzungen an den Tests erfordert. Java -> Declaration Redundancy)
RNr. 15	Speichern von Berechnungsergebnissen, die häufiger als einmal verwendet werden, in Variablen	(Wiederholte Berechnungen desselben Werts innerhalb desselben Scopes lassen sich weder per Regex noch mit der Code-Inspection-Funktion in IntelliJ ermitteln. Daher wird während des übrigen Refactorings darauf geachtet, ob solche Strukturen auffallen. Wenn dem so ist, werden sie für RNr. 15 vorgemerkt.)
RNr. 16	Vermeidung der Datentypen byte, short, float, double und char	*^(?!(\s**)).*\b(byte|short|float|double|char)\b*
RNr. 17	Umwandlung von statischen Variablen in nicht-statische	*^(?!(\s**)|import).*\bstatic\b(?! (void|class|(.+\{)))*
RNr. 18	Verwendung primitiver Datentypen (Ausnahme bei Speicherung in Datenstrukturen)	*^(?!(\s**)).*\b(Integer|Byte|Long|Double|Short|Float|Boolean|Char)\b*
RNr. 19	Verwendung von + zur String-Konkatenation	*StringBuffer|StringBuilder*

A.2 Messergebnisse

Je gemessener Einheit ist für jeden Code-Zustand der niedrigste Wert grün unterlegt.

A.2.1 GraphStream: *gs-algo*

	Zeit (Sek.)	Energie (Joule)
Ausgangszustand	94,833404	1504,490967
	95,14559	1500,297424
	95,295434	1494,76001
	94,209516	1472,737976
	92,854123	1453,140991
	91,811438	1439,994446
	92,279914	1443,110779
	93,825336	1449,343323
	91,991139	1428,591919
	93,447858	1474,710327
Durchschnitt	**93,5693752**	**1466,117816**
Design	92,209911	1433,058533
	91,894232	1429,966675
	93,244203	1466,973328
	90,275423	1426,703918
	90,698408	1404,779114
	89,852062	1410,830688
	90,453451	1398,558289
	89,487957	1402,296143
	91,49549	1431,008057
	94,190448	1490,712708
Durchschnitt	**91,3801585**	**1429,488745**
Kontrollstruktu-ren	92,345634	1417,866943
	91,445843	1430,608643
	91,776445	1437,156189
	92,272126	1451,824036
	94,540666	1478,770325
	92,285575	1457,271057
	93,002943	1441,578003
	91,99268	1438,599365

	92,670835	1460,530457
	92,828802	1453,13623
Durchschnitt	**92,5161549**	**1446,734125**
Datenstrukturen	92,868889	1413,007996
	90,546115	1412,366211
	91,769912	1404,86615
	90,51206	1395,786438
	92,983297	1440,557495
	88,804422	1395,803711
	91,416781	1411,32251
	89,743594	1397,055664
	92,168179	1426,104431
	91,632164	1414,682373
Durchschnitt	**91,2445413**	**1411,155298**
Energy Smells A	93,083639	1435,060974
	93,607141	1462,855164
	91,518508	1439,036804
	92,056992	1449,826904
	92,765411	1443,623291
	93,653089	1464,479187
	93,422477	1454,2323
	93,369942	1463,182739
	92,329353	1412,945129
	91,68438	1439,874573
Durchschnitt	**92,7490932**	**1446,511707**
Energy Smells B	92,474646	1399,536865
	91,075293	1426,987427
	88,980336	1409,434326
	90,090835	1409,791016
	90,691264	1408,310913
	92,105048	1417,477356
	90,644462	1423,273132
	91,189581	1417,158936
	91,729143	1428,758179
	90,727298	1423,925598
Durchschnitt	**90,9707906**	**1416,465375**
Endzustand ohne ESA	91,139076	1382,450745
	92,874041	1443,211853

	94,765703	1458,940918
	92,347041	1448,823669
	92,989165	1443,619751
	90,570075	1413,62146
	91,797354	1423,802734
	91,914423	1440,069763
	92,659126	1443,360229
	90,762444	1408,459106
Durchschnitt	**92,1818448**	**1430,636023**
Endzustand mit ESA	93,491868	1459,93811
	94,017906	1473,584534
	91,811066	1421,983765
	92,90641	1428,002686
	92,870779	1431,88446
	92,738097	1436,08252
	93,779293	1452,25946
	93,628754	1457,584717
	93,47827	1444,65741
	92,619097	1441,232483
Durchschnitt	**93,134154**	**1444,721015**

A.2.2 LanguageTool: *languagetool-core.rules*

	Zeit (Sek.)	Energie (Joule)
Ausgangszustand	142,168484	2193,472473
	141,078696	2185,85144
	142,825746	2214,95105
	142,149988	2198,59967
	141,463176	2215,5849
	139,696713	2194,867371
	138,095406	2104,998352
	139,619155	2160,563171
	139,175812	2155,139404
	138,088766	2172,949524
Durchschnitt	**140,4361942**	**2179,697736**
Design	140,641819	2149,229919

	142,475353	2190,913635
	138,868659	2181,552979
	140,401742	2161,763306
	138,882798	2153,807739
	141,750219	2175,50708
	142,12249	2191,163757
	141,538069	2168,500732
	141,500863	2191,762939
	140,527779	2135,262817
Durchschnitt	**140,8709791**	**2169,94649**
Kontrollstruktu-ren	137,322514	2153,820618
	137,338938	2151,786011
	138,352712	2112,957947
	137,527155	2109,359436
	137,562276	2147,013428
	140,179386	2188,586853
	139,923648	2190,443542
	139,63537	2163,079895
	141,441187	2181,394714
	139,311411	2146,096985
Durchschnitt	**138,8594597**	**2154,453943**
Datenstrukturen	142,307909	2142,858704
	139,800771	2173,585999
	138,537806	2169,236145
	141,483166	2131,836365
	138,694602	2174,625183
	138,661718	2156,233215
	138,606312	2163,290588
	138,745111	2180,695313
	135,748024	2138,487244
	138,648361	2145,177795
Durchschnitt	**139,123378**	**2157,602655**
Energy Smells A	137,110674	2070,465332
	137,086782	2114,553101
	136,817682	2129,531067
	135,897911	2142,623291
	137,505927	2161,150146
	139,077748	2147,524231

	137,487308	2078,179993
	135,553067	2132,86145
	136,023763	2131,060791
	135,472317	2118,727295
Durchschnitt	**136,8033179**	**2122,66767**
Energy Smells B	135,169852	2100,080566
	135,917202	2141,669983
	136,662438	2115,8349
	135,712499	2142,044983
	136,137833	2147,767334
	135,584691	2085,357727
	135,726572	2134,534912
	137,602396	2148,956909
	133,997494	2090,618896
	135,495702	2130,669189
Durchschnitt	**135,8006679**	**2123,75354**
Endzustand ohne ESB	140,693203	2181,599548
	139,181054	2183,077332
	139,715481	2182,808594
	139,634969	2159,201477
	138,451011	2179,465515
	139,748618	2170,192322
	138,863559	2145,162659
	139,282417	2184,307983
	140,172761	2196,508789
	139,367323	2132,333618
Durchschnitt	**139,5110396**	**2171,465784**

A.3 Statistische Analyse – GraphStream: *gs-algo*

A.3.1 Energie (Joule)

Deskriptive Statistik

	N Analysis	N Missing	Mean	Standard Deviation	SE of Mean
Ausgangszustand	10	0	1466,11782	27,17946	8,5949
Design	10	0	1429,48875	29,49825	9,32817
Kontrollstrukturen	10	0	1446,73412	17,23182	5,44918
Datenstrukturen	10	0	1411,15530	14,20313	4,49142
Energy Smells A	10	0	1446,51171	16,00196	5,06026
Energy Smells B	10	0	1416,46537	9,50079	3,00441
Endzustand ohne ESA	10	0	1430,63602	23,23650	7,34803
Endzustand mit ESA	10	0	1444,72101	16,10737	5,0936

ANOVA

	DF	Sum of Squares	Mean Square	F Value	Prob>F
Model	7	22770,52138	3252,93163	7,99923	<0.0001
Error	72	29279,19424	406,65548		
Total	79	52049,71562			

Tukey-Test

	MeanDiff	SEM	q Value	Prob	Alpha	Sig	LCL	UCL
Design Ausgangszustand	-36,62907	9,01838	5,74398	0,00295	0,05	1	-64,78296	-8,47519
Kontrollstrukturen Ausgangszustand	-19,38369	9,01838	3,03965	0,39437	0,05	0	-47,53758	8,77019
Kontrollstrukturen Design	17,24538	9,01838	2,70433	0,54728	0,05	0	-10,90851	45,39927
Datenstrukturen Ausgangszustand	-54,96252	9,01838	8,61893	<0.0001	0,05	1	-83,11640	-26,80863
Datenstrukturen Design	-18,33345	9,01838	2,87495	0,46768	0,05	0	-46,48733	9,82044
Datenstrukturen Kontrollstrukturen	-35,57883	9,01838	5,57928	0,00433	0,05	1	-63,73271	-7,42494
Energy Smells A Ausgangszustand	-19,60611	9,01838	3,07453	0,37950	0,05	0	-47,76000	8,54778
Energy Smells A Design	17,02296	9,01838	2,66945	0,56376	0,05	0	-11,13092	45,17685
Energy Smells A Kontrollstrukturen	-0,22242	9,01838	0,03488	1,00000	0,05	0	-28,37630	27,93147

Energy Smells A Datenstrukturen	35,35641	9,01838	5,54440	0,00469	0,05	1	7,20252	63,51029
Energy Smells B Ausgangszustand	-49,65244	9,01838	7,78623	<0.0001	0,05	1	-77,80633	-21,49856
Energy Smells B Design	-13,02337	9,01838	2,04226	0,83347	0,05	0	-41,17726	15,13052
Energy Smells B Kontrollstrukturen	-30,26875	9,01838	4,74658	0,02629	0,05	1	-58,42264	-2,11486
Energy Smells B Datenstrukturen	5,31008	9,01838	0,83270	0,99892	0,05	0	-22,84381	33,46396
Energy Smells B Energy Smells A	-30,04633	9,01838	4,71171	0,02819	0,05	1	-58,20022	-1,89245
Endzustand ohne ESA Ausgangszustand	-35,48179	9,01838	5,56407	0,00448	0,05	1	-63,63568	-7,32791
Endzustand ohne ESA Design	1,14728	9,01838	0,17991	1,00000	0,05	0	-27,00661	29,30116
Endzustand ohne ESA Kontrollstrukturen	-16,09810	9,01838	2,52442	0,63201	0,05	0	-44,25199	12,05578
Endzustand ohne ESA Datenstrukturen	19,48072	9,01838	3,05486	0,38785	0,05	0	-8,67316	47,63461
Endzustand ohne ESA Energy Smells A	-15,87568	9,01838	2,48954	0,64821	0,05	0	-44,02957	12,27820
Endzustand ohne ESA Energy Smells B	14,17065	9,01838	2,22217	0,76548	0,05	0	-13,98324	42,32453
Endzustand mit ESA Ausgangszustand	-21,39680	9,01838	3,35533	0,27060	0,05	0	-49,55069	6,75708
Endzustand mit ESA Design	15,23227	9,01838	2,38864	0,69416	0,05	0	-12,92162	43,38615
Endzustand mit ESA Kontrollstrukturen	-2,01311	9,01838	0,31569	1,00000	0,05	0	-30,16700	26,14078
Endzustand mit ESA Datenstrukturen	33,56572	9,01838	5,26360	0,00883	0,05	1	5,41183	61,71960
Endzustand mit ESA Energy Smells A	-1,79069	9,01838	0,28081	1,00000	0,05	0	-29,94458	26,36319
Endzustand mit ESA Energy Smells B	28,25564	9,01838	4,43090	0,04853	0,05	1	0,10175	56,40953
Endzustand mit ESA Endzustand ohne ESA	14,08499	9,01838	2,20873	0,77092	0,05	0	-14,06889	42,23888

A.3.2 Zeit (Sekunden)

Deskriptive Statistik

	N Analysis	N Missing	Mean	Standard Deviation	SE of Mean
Ausgangszustand	10	0	93,56938	1,30414	0,41241
Design	10	0	91,38016	1,52016	0,48072
Kontrollstrukturen	10	0	92,51615	0,85413	0,2701
Datenstrukturen	10	0	91,24454	1,34173	0,42429
Energy Smells A	10	0	92,74909	0,8033	0,25403
Energy Smells B	10	0	90,97079	1,00897	0,31906
Endzustand ohne ESA	10	0	92,18184	1,2464	0,39415
Endzustand mit ESA	10	0	93,13415	0,66518	0,21035

ANOVA

	DF	Sum of Squares	Mean Square	F Value	Prob>F
Model	7	62,42948	8,9185	6,9944	<0.0001
Error	72	91,80654	1,27509		
Total	79	154,23601			

Tukey-Test

	MeanDiff	SEM	q Value	Prob	Alpha	Sig	LCL	UCL
Design Ausgangszustand	-2,18922	0,50499	6,13081	0,00116	0,05	1	-3,76572	-0,61271
Kontrollstrukturen Ausgangszustand	-1,05322	0,50499	2,9495	0,43393	0,05	0	-2,62973	0,52329
Kontrollstrukturen Design	1,136	0,50499	3,18131	0,3357	0,05	0	-0,44051	2,7125
Datenstrukturen Ausgangszustand	-2,32483	0,50499	6,51061	4,45E-04	0,05	1	-3,90134	-0,74833
Datenstrukturen Design	-0,13562	0,50499	0,37979	0,99999	0,05	0	-1,71212	1,44089
Datenstrukturen Kontrollstrukturen	-1,27161	0,50499	3,5611	0,2047	0,05	0	-2,84812	0,30489
Energy Smells A Ausgangszustand	-0,82028	0,50499	2,29717	0,73418	0,05	0	-2,39679	0,75622
Energy Smells A Design	1,36893	0,50499	3,83365	0,13613	0,05	0	-0,20757	2,94544
Energy Smells A Kontrollstrukturen	0,23294	0,50499	0,65233	0,99978	0,05	0	-1,34357	1,80944
Energy Smells A Datenstrukturen	1,50455	0,50499	4,21344	0,07211	0,05	0	-0,07195	3,08106
Energy Smells B Ausgangszustand	-2,59858	0,50499	7,27723	<0.0001	0,05	1	-4,17509	-1,02208
Energy Smells B Design	-0,40937	0,50499	1,14642	0,992	0,05	0	-1,98587	1,16714
Energy Smells B Kontrollstrukturen	-1,54536	0,50499	4,32773	0,05872	0,05	0	-3,12187	0,03114
Energy Smells B Datenstrukturen	-0,27375	0,50499	0,76663	0,99937	0,05	0	-1,85026	1,30276
Energy Smells B Energy Smells A	-1,7783	0,50499	4,98007	0,01627	0,05	1	-3,35481	-0,2018
Endzustand ohne ESA Ausgangszustand	-1,38753	0,50499	3,88572	0,12533	0,05	0	-2,96404	0,18898
Endzustand ohne ESA Design	0,80169	0,50499	2,24509	0,75608	0,05	0	-0,77482	2,37819
Endzustand ohne ESA Kontrollstrukturen	-0,33431	0,50499	0,93622	0,99771	0,05	0	-1,91082	1,2422
Endzustand ohne ESA Datenstrukturen	0,9373	0,50499	2,62488	0,58481	0,05	0	-0,6392	2,51381
Endzustand ohne ESA Energy Smells A	-0,56725	0,50499	1,58856	0,94956	0,05	0	-2,14375	1,00926
Endzustand ohne ESA Energy Smells B	1,21105	0,50499	3,39151	0,25812	0,05	0	-0,36545	2,78756
Endzustand mit ESA Ausgangszustand	-0,43522	0,50499	1,21882	0,98848	0,05	0	-2,01173	1,14128
Endzustand mit ESA Design	1,754	0,50499	4,912	0,01875	0,05	1	0,17749	3,3305
Endzustand mit ESA Kontrollstrukturen	0,618	0,50499	1,73068	0,92216	0,05	0	-0,95851	2,1945
Endzustand mit ESA Datenstrukturen	1,88961	0,50499	5,29179	0,0083	0,05	1	0,31311	3,46612
Endzustand mit ESA Energy Smells A	0,38506	0,50499	1,07835	0,99448	0,05	0	-1,19145	1,96157
Endzustand mit ESA Energy Smells B	2,16336	0,50499	6,05841	0,00138	0,05	1	0,58686	3,73987
Endzustand mit ESA Endzustand ohne ESA	0,95231	0,50499	2,6669	0,56496	0,05	0	-0,6242	2,52882

A.4 Statistische Analyse – LanguageTool: *languagetool-core.rules*

A.4.1 Energie (Joule)

Deskriptive Statistik

	N Analysis	N Missing	Mean	Standard Deviation	SE of Mean
Ausgangszustand	10	0	2179,69774	33,23398	10,50951
Design	10	0	2169,94649	19,70062	6,22988
Kontrollstrukturen	10	0	2154,45394	28,18327	8,91233
Datenstrukturen	10	0	2157,60266	17,15657	5,42539
Energy Smells A	10	0	2122,66767	28,87531	9,13117
Energy Smells B	10	0	2123,75354	24,0988	7,62071
Endzustand ohne ESB	10	0	2171,46578	19,99296	6,32233

ANOVA

	DF	Sum of Squares	Mean Square	F Value	Prob>F
Model	6	31291,0606	5215,17676	8,30535	<0.0001
Error	63	39559,5951	627,93008		
Total	69	70850,6556			

Tukey-Test

	MeanDiff	SEM	q Value	Prob	Alpha	Sig	LCL	UCL
Design Ausgangszustand	-9,75125	11,20652	1,23056	0,9757	0,05	0	-43,88205	24,37956
Kontrollstrukturen Ausgangszustand	-25,24379	11,20652	3,18566	0,28332	0,05	0	-59,3746	8,88701
Kontrollstrukturen Design	-15,49255	11,20652	1,95509	0,80899	0,05	0	-49,62335	18,63826
Datenstrukturen Ausgangszustand	-22,09508	11,20652	2,7883	0,44255	0,05	0	-56,22589	12,03572
Datenstrukturen Design	-12,34384	11,20652	1,55774	0,92536	0,05	0	-46,47464	21,78697
Datenstrukturen Kontrollstrukturen	3,14871	11,20652	0,39735	0,99996	0,05	0	-30,98209	37,27952
Energy Smells A Ausgangszustand	-57,03007	11,20652	7,19695	<0.0001	0,05	1	-91,16087	-22,89926
Energy Smells A Design	-47,27882	11,20652	5,96638	0,00149	0,05	1	-81,40963	-13,14802
Energy Smells A Kontrollstrukturen	-31,78627	11,20652	4,01129	0,08397	0,05	0	-65,91708	2,34453
Energy Smells A Datenstrukturen	-34,93499	11,20652	4,40864	0,04148	0,05	1	-69,06579	-0,80418

Energy Smells B	Ausgangszustand	-55,9442	11,20652	7,05991	<0.0001	0,05	1	-90,075	-21,81339
Energy Smells B	Design	-46,19295	11,20652	5,82935	0,00206	0,05	1	-80,32376	-12,06215
Energy Smells B	Kontrollstrukturen	-30,7004	11,20652	3,87426	0,1053	0,05	0	-64,83121	3,4304
Energy Smells B	Datenstrukturen	-33,84912	11,20652	4,27161	0,05332	0,05	0	-67,97992	0,28169
Energy Smells B	Energy Smells A	1,08587	11,20652	0,13703	1	0,05	0	-33,04493	35,21668
Endzustand ohne ESB	Ausgangszustand	-8,23195	11,20652	1,03884	0,98984	0,05	0	-42,36276	25,89885
Endzustand ohne ESB	Design	1,51929	11,20652	0,19173	1	0,05	0	-32,61151	35,6501
Endzustand ohne ESB	Kontrollstrukturen	17,01184	11,20652	2,14682	0,73307	0,05	0	-17,11896	51,14265
Endzustand ohne ESB	Datenstrukturen	13,86313	11,20652	1,74947	0,87686	0,05	0	-20,26768	47,99393
Endzustand ohne ESB	Energy Smells A	48,79811	11,20652	6,15811	9,43E-04	0,05	1	14,66731	82,92892
Endzustand ohne ESB	Energy Smells B	47,71224	11,20652	6,02108	0,00131	0,05	1	13,58144	81,84305

A.4.2 Zeit (Sekunden)

Deskriptive Statistik

	N Analysis	N Missing	Mean	Standard Deviation	SE of Mean
Ausgangszustand	10	0	140,43619	1,7291	0,54679
Design	10	0	140,87098	1,25201	0,39592
Kontrollstrukturen	10	0	138,85946	1,4428	0,45625
Datenstrukturen	10	0	139,12338	1,79312	0,56704
Energy Smells A	10	0	136,80332	1,10914	0,35074
Energy Smells B	10	0	135,80067	0,93799	0,29662
Endzustand ohne ESB	10	0	139,51104	0,6389	0,20204

ANOVA

	DF	Sum of Squares	Mean Square	F Value	Prob>F
Model	6	205,57074	34,26179	19,38436	<0.0001
Error	63	111,35229	1,7675		
Total	69	316,92304			

Tukey-Test

	MeanDiff	SEM	q Value	Prob	Alpha	Sig	LCL	UCL
Design Ausgangszustand	0,43478	0,59456	1,03418	0,99008	0,05	0	-1,37601	2,24558
Kontrollstrukturen Ausgangszustand	-1,57673	0,59456	3,75041	0,12819	0,05	0	-3,38753	0,23406
Kontrollstrukturen Design	-2,01152	0,59456	4,78459	0,02003	0,05	1	-3,82232	-0,20072
Datenstrukturen Ausgangszustand	-1,31282	0,59456	3,12266	0,30606	0,05	0	-3,12361	0,49798
Datenstrukturen Design	-1,7476	0,59456	4,15684	0,06539	0,05	0	-3,5584	0,0632
Datenstrukturen Kontrollstrukturen	0,26392	0,59456	0,62776	0,99938	0,05	0	-1,54688	2,07472
Energy Smells A Ausgangszustand	-3,63288	0,59456	8,64115	<0.0001	0,05	1	-5,44368	-1,82208
Energy Smells A Design	-4,06766	0,59456	9,67532	<0.0001	0,05	1	-5,87846	-2,25686
Energy Smells A Kontrollstrukturen	-2,05614	0,59456	4,89073	0,01616	0,05	1	-3,86694	-0,24534
Energy Smells A Datenstrukturen	-2,32006	0,59456	5,51849	0,00419	0,05	1	-4,13086	-0,50926
Energy Smells B Ausgangszustand	-4,63553	0,59456	11,02604	<0.0001	0,05	1	-6,44633	-2,82473
Energy Smells B Design	-5,07031	0,59456	12,06022	<0.0001	0,05	1	-6,88111	-3,25951
Energy Smells B Kontrollstrukturen	-3,05879	0,59456	7,27563	<0.0001	0,05	1	-4,86959	-1,24799
Energy Smells B Datenstrukturen	-3,32271	0,59456	7,90339	<0.0001	0,05	1	-5,13351	-1,51191
Energy Smells B Energy Smells A	-1,00265	0,59456	2,3849	0,62745	0,05	0	-2,81345	0,80815
Endzustand ohne ESB Ausgangszustand	-0,92515	0,59456	2,20057	0,71008	0,05	0	-2,73595	0,88564
Endzustand ohne ESB Design	-1,35994	0,59456	3,23475	0,26635	0,05	0	-3,17074	0,45086
Endzustand ohne ESB Kontrollstrukturen	0,65158	0,59456	1,54985	0,92704	0,05	0	-1,15922	2,46238
Endzustand ohne ESB Datenstrukturen	0,38766	0,59456	0,92209	0,99463	0,05	0	-1,42314	2,19846
Endzustand ohne ESB Energy Smells A	2,70772	0,59456	6,44058	4,73E-04	0,05	1	0,89692	4,51852
Endzustand ohne ESB Energy Smells B	3,71037	0,59456	8,82548	<0.0001	0,05	1	1,89957	5,52117

Erklärung zur selbstständigen Bearbeitung einer Abschlussarbeit

Hiermit versichere ich, dass ich die vorliegende Arbeit ohne fremde Hilfe selbständig verfasst und nur die angegebenen Hilfsmittel benutzt habe. Wörtlich oder dem Sinn nach aus anderen Werken entnommene Stellen sind unter Angabe der Quellen kenntlich gemacht. Ich versichere, dass ich textgenerierende KI-Tools lediglich im Schreibprozess sprachunterstützend genutzt habe und in der vorliegenden Arbeit mein gestalterischer Einfluss überwiegt.

_____	_____	_____
Ort	Datum	Unterschrift im Original