HAW HAMBURG

BACHELORARBEIT

Entwicklung eines Werkzeugs zur Reduzierung des Energieverbrauchs von Websites durch statische Code-Analyse

vorgelegt am 12. Oktober 2024

Leon Beitz

Erstprüfer: Prof. Dr.-Ing. Lars Hamann
Zweitprüferin: Prof. Dr. Bettina Buth

HOCHSCHULE FÜR ANGEWANDTE WISSENSCHAFTEN HAMBURG
Department Informatik
Berliner Tor 7
20099 Hamburg

Zusammenfassung

Für diese Arbeit[1] wurde eine Website entwickelt, auf der Quellcode eingesendet werden kann, um diesen auf Probleme der Energieeffizienz zu überprüfen. Durch die stetig steigende Komplexität von Webanwendungen und die Klimakrise bekommmt energieeffizientes Webdesign eine immer größere Bedeutung. Diese Arbeit beschäftigt sich mit Konzepten des *Sustainable Webdesigns* und der Energieeffizienz von Websites. Die Analyse des Quellcodes erfolgt mithilfe von Regeln, die aus den *Web Sustainability Guidelines* abgeleitet wurden. Diese Regeln prüfen den Quellcode auf ineffiziente Strukturen und schlagen Maßnahmen zur Verbesserung vor. Die Regeln analysieren den Quellcode mithilfe von statischer Programmanalyse. Das Werkzeug wurde in der Programmiersprache Rust entwickelt und soll Entwickler dabei unterstützen, energieeffizienten Code zu schreiben.

Abstract

For this thesis, a website was developed on which source code can be submitted to check it for energy efficiency issues. Due to the increasing complexity of web applications and the climate crisis, energy-efficient web design is becoming increasingly important. This thesis deals with concepts of sustainable webdesign and the energy efficiency of websites. The analysis of the source code is done using rules derived from the Web Sustainability Guidelines. These rules check the source code for inefficient structures and suggest measures for improvement. The rules analyze the source code using static program analysis. The tool was developed in the programming language Rust and is intended to help developers write energy-efficient code.

[1] Zur besseren Lesbarkeit wird in dieser Arbeit das generische Maskulinum genutzt und auf die Verwendung von weiblichen und männlichen Sprachformen verzichtet. Sämtliche Personenbezeichnungen gelten gleichermaßen, sofern nicht anders kenntlich gemacht, für alle Geschlechter.

Inhaltsverzeichnis

Abbildungsverzeichnis

Tabellenverzeichnis

1 Theoretische Grundlagen

Im folgenden Kapitel werden die theoretischen Grundlagen erläutert, die für das Verständnis dieser Arbeit notwendig sind. Es wird mit dem nachhaltigen Webdesign begonnen, um die Motivation für die Entwicklung des Werkzeugs und die Hintergründe zu erläutern.

1.1 Nachhaltiges Webdesign

In den letzten Jahren ist die Bewegung für ein nachhaltiges Webdesign immer stärker geworden. Dies hat, nicht zuletzt durch die Klimakrise, die auf die Dringlichkeit des Klimawandels aufmerksam gemacht hat, an Bedeutung gewonnen. Internetgeschwindigkeiten und Rechenleistungen haben sich in den letzten Jahren stark verbessert, was dazu geführt hat, dass Webdesigner und Entwickler immer komplexere Websites erstellen können. Durch die steigende Komplexität steigt auch der Energieverbrauch, der durch das Aufrufen und Nutzen von Websites entsteht. Die meisten Webentwickler sind sich jedoch nicht bewusst, dass ihre Websites einen Beitrag zu den weltweiten CO2-Emissionen leisten. Nachhaltiges Webdesign strebt an, dass Webentwickler bewusster mit dem Energieverbrauch ihrer Websites umgehen und diesen reduzieren. Sie sollen Verantwortung übernehmen, Webdienste zu schaffen, die gut für Mensch und Umwelt sind. Nachhaltiges Webdesign ist ein Ansatz, bei dem die Umweltauswirkungen bei der Entwicklung von Websites im Vordergrund steht. Es geht also darum, den Energieverbrauch und die daraus resultierenden CO2-Emissionen zu reduzieren [1, S. 9 - 10].

1.1.1 Web Sustainability Guidelines

Vom *World Wide Web Consotrium (W3C)* [2] gibt es seit dem 7. September 2023 Richtlinien für nachhaltiges Webdesign. Die *Web Sustainability Guidelines (WSG)* [3] wurden von der *Sustainable Web Design Community Group* [4], einer über die W3C organisierte Gruppe, entwickelt. Sie sind kein *W3C Standard*[1] und streben auch nicht an, einer zu werden. Des

[1] „W3C publishes documents that define Web technologies. These documents follow a process designed to promote consensus, fairness, public accountability, and quality. At the end of this process, W3C publishes Recommendations, which are considered Web standards.“ [5]

Weiteren befinden sich die *WSG* im ständigen Wandel und können sich jederzeit ändern. *WSG* ist darauf ausgelegt, auf Webtechnologien anwendbar und mit einer Kombination aus automatisierten Tests und menschlicher Bewertung testbar zu sein. Außerdem orientieren sich die *WSG* an den *environmental, social, and (corporate) governance (ESG)*-Prinzipien [6]. *ESG* basiert auf dem *Triple-Bottom-Line-Ansatz* [7], der von *John Elkington* entwickelt wurde. Die *Triple-Bottom-Line* ist ein Konzept, die drei *P's* (People, Planet, Profit) in den Vordergrund zu stellen. Die *ESG*-Prinzipien definieren Nachhaltigkeit auf Basis der drei Säulen: Umwelt, Soziales und Wirtschaft. Es geht also nicht nur um den Naturschutz, sondern auch um die sozialen und wirtschaftlichen Auswirkungen von Webdiensten.

1.1.2 Ziele

Die Ziele für die *WSG*, und damit für nachhaltiges Websdesign, basieren auf folgenden Prinzipien [8]:

- **Sauberkeit:** Das Internet sollte auf saubere, erneuerbare Energiequellen umgestellt werden. Dies ist besonders für Rechenzentren und die Infrastruktur des Internets wichtig, da diese in der Verantwortung der Entwickler und Betreiber liegen.
- **Effizienz:** Effizienz im Rahmen des nachhaltigen Webdesigns bedeutet, dass Websites effizient sind und den Energieverbrauch möglichst gering halten. Bessere Internetgeschwindigkeiten und Rechenleistungen verleiten dazu, weniger effizient zu arbeiten, da die Hardware die ineffiziente Software ausgleichen kann.
- **Offenheit:** Hier geht es darum, Projekte offenzulegen und zu teilen. Dies ermöglicht es, andere Entwickler von den Erfahrungen profitieren zu lassen und die eigenen Projekte zu verbessern. Ebenfalls geht auch um Barrierefreiheit, um allen Menschen den Zugang zu Webdiensten und Informationen zu ermöglichen.
- **Ehrlichkeit:** Dieser Aspekt bezieht sich auf *Greenwashing* und diesem entgegenzuwirken. Es soll klar kommuniziert werden, welche Maßnahmen getroffen wurden, um die Website nachhaltiger zu gestalten. Dies passiert nicht im Sinne von Marketing, sondern um die Nutzer zu informieren.
- **Regeneration:** Um Websites zu schaffen die nicht lediglich Schadensbgrenzung betreiben, ist das Prinzip der Regeneration von Bedeutung. Der Planet soll aktiv regeneriert werden. Dies soll mittels *wiedergutmachenden Technologien* (engl. Orig. *redemptive technologies*), die in *The Real World of Technology* [9] definiert werden, erreicht werden. Beispiele, wie dies gelingen kann, sind z.B. die *Ecosia*[2] oder *Do Nation*[3].

[2] eine Internetsuche, bei der der gesamte Profit in Klimaschutzprojekte investiert wird (https://www.ecosia.org)
[3] eine Website die dabei hilft klimafreundliche Gewohnheiten einzuüben (https://wearedonation.com)

- **Resilient:** Die Resilienz von Webdiensten zielt auf die Widerstandsfähigkeit gegenüber Störungen ab. Internetseiten sollten auch in Krisenfällen, Naturkatastrophen, Kriegen oder Pandemien erreichbar sein. Dies gelingt, indem *Single-Points-of-Failure* vermieden werden und Websites leichtgewichtig sind. So können sie auch mit schlechtem Internet und langsamen Geräten aufgerufen werden. Mit dem Endziel, das Internet und die hier verteilten Informationen für alle Menschen zugänglich zu machen.

1.1.3 Umsetzung

Die *WSG* geben konkrete Methoden an, um die Prinzipien des nachhaltigen Webdesigns umzusetzen [3]. Diese Methoden unterteilen sich in vier Kategorien:

- **User-Experience Design**
  - Nutzer- und Produktanforderungen erfassen und regelmäßig testen
  - Informationen für leichte Zugänglichkeit (*ease of access*) optimieren
  - angemessene und möglichst leichtgewichtige Technologien verwenden
- **Web Development**
  - Code für Zugänglichkeit, Performance, Sicherheit und andere ESG-Ziele optimieren
  - Lösungen nicht von leistungsstarker Hardware abhängig machen
  - technische Features bereitstellen, die die digitale Nachhaltigkeit erhöhen
- **Hosting, Infrastructure and Systems**
  - nachhaltige Anbieter für Hosting, CDNs und Drittanbieter von Lösungen finden
  - Einstellungen für Caching, Komprimierung und Fehlerbehandlung vom Backend aus verwalten
- **Buisness Strategy and Product Management**
  - ESG-Ziele in die Produktentwicklung integrieren
  - öffentlich über Nachhaltigkeits-Bemühungen informieren und berichten
  - Veränderungen innerhalb eines Unternehmens und seiner Mitarbeiter beeinflussen, um ESG-Ziele zu erreichen

Diese Kategorien werden in viele Unterpunkte mit konkreten Vorgaben und testbaren Erfolgskriterien unterteilt. Die Unterpunkte werden dann, in Bezug auf Aufwand und Auswirkung, in drei Kategorien eingeteilt: niedrig, mittel und hoch [3].
Es wird sich in dieser Arbeit auf die Kategorien *User-Experience Design* und *Web Development* konzentriert. *User-Experience Design* und *Web Development* sind die beiden Kategorien, die auf Code-Ebene stattfinden.
Nachhaltiges Webdesign ist also ein Ansatz, um den Energieverbrauch und die daraus resultierenden CO2-Emissionen von Websites zu reduzieren.

1.2 Energieverbrauch von Websites messen

Im Folgenden geht es um die Reduzierung des Energieverbrauchs und die Effizienzsteigerung von Websites. Dazu ist es notwendig, zu verstehen, wie sich der Energieverbrauch von Websites zusammensetzt. Das Internet verursacht jährliche Emissionen, die vergleichbar mit den Emissionen Deutschlands sind. Hypothetisch ist das Internet damit eins der sechs Länder mit den höchsten CO2-Emissionen der Welt [10]. Laut einem im *Journal of Cleaner Production* veröffentlichten Paper, wird der Stromverbrauch von Kommunikationstechnologien bis 2040 auf 14 Prozent des weltweiten Energieverbrauchs steigen [11].

1.2.1 Entwicklung des Energieverbrauchs

Tabelle 1.1: Globaler Energieverbrauch durch das Internet 2015 und 2022 [12]

Description	**2015**	**2022**	**Change**
Internet users	3 billion	5.3 billion	+78%
Internet traffic	0.6 ZB	4.4 ZB	+600%
Data centre workloads	180 million	800 million	+340%
Data centre energy use (excluding crypto)	200 TWh	240-340 TWh	+20-70%
Crypto mining energy use	4 TWh	100-150 TWh	+2300-3500%
Data transmission network energy use	220 TWh	260-360 TWh	+18-64%

In der Tabelle 1.1 ist zu sehen, dass der Datenverkehr im Internet von 2015 bis 2022 um 600% gestiegen ist. Die Internetnutzer und der Energieverbrauch der Infrastruktur sind zwar auch gestiegen, aber nicht in dem Maße, in dem es der Datenverkehr vermuten lässt. Dies liegt daran, dass die Effizienz der Infrastruktur sich rapide verbessert hat [12]. Auch die Datenzentren sind effizienter geworden. Der Energieverbrauch ist nur um 20-70% gestiegen, obwohl die Rechenlasten um 340% gestiegen sind. Die *International Energy Agency* (IEA) hat Erhebungen zum Energieverbrauch der Haushalte gemacht. Hier ist der Energieverbrauch zwischen 2015

und 2018 um 1% gestiegen. Durch die gesteigerte Effizienz ist der Energieverbrauch hier weniger stark gestiegen als zu erwarten wäre. Wie im *Journal of Cleaner Production* veröffentlicht, wird der Energieverbrauch und Bedarf an Technologien weiter steigen [11]. Demnach ist es wichtig, dass Webentwickler und Designer sich mit dem Energieverbrauch ihrer Websites auseinandersetzen und diesen reduzieren.

1.2.2 Bestandteile des Energieverbrauchs

Um den Energieverbrauch von Websites zu reduzieren, muss festgestellt werden, aus welchen Bestandteilen der Energieverbrauch besteht. Der Energieverbrauch von Websites setzt sich aus dem Energieverbrauch des Servers, des Netzwerks und des Endgeräts zusammen. Server stehen hier in der Regel für Rechenzentren, die die Website hosten. Rechenzentren sind für 3% des weltweiten Stromverbrauchs verantwortlich, was 416,2 Terawattstunden entspricht und somit den Stromverbrauch des Vereinigten Königreichs (ca. 300 Terawattstunden) um Längen übertrifft [13]. Darüber hinaus tragen Rechenzentren zu 2% der globalen CO2-Emissionen bei, was mehr ist als die gesamten Emissionen des internationalen Flugverkehrs [13]. Der zweite Energieverbraucher ist die Infrastruktur des Internets, also das Netzwerk, das die Daten zwischen Servern und Endgeräten überträgt. Dritter Energieverbraucher ist das Endgerät, auf dem die Website aufgerufen wird.

1.2.3 CO2-Emissionen und Energieverbrauch

CO2-Emissionen und der Energieverbrauch von Websites sind eng miteinander verknüpft. Ein höherer Energieverbrauch führt zu höheren CO2-Emissionen. Die CO2-Emissionen von Websites sind schwer zu bestimmen, da sie von vielen Faktoren abhängen. In den meisten Fällen wird der Datenverkehr allein als Merkmal für den gesamten Energieverbrauch und die CO2-Emissionen verwendet. Hier spielen die Datenübertragung und die CO2-Intensität des Stroms, der dafür verwendet wird, eine Rolle.
Die Energieeffizienz der Datenübertragung wird in den meisten Fällen in kWh / GB[4] gemessen. Diese fungiert als guter Anhaltspunkt für den gesamten Energieverbrauch, welcher auf der Seite des Servers, des Netzwerks und des Endgeräts entsteht [1, S. 32]. Je höher die Datenübertragung, desto höher der Energieverbrauch bei allen Beteiligten. Daten müssen aufbereitet, versendet und auf der Empfängerseite wieder verarbeitet werden. Wie hoch der Energieverbrauch genau ist, variiert stark zwischen den unterschiedlichen Quellen, die verwendet werden und liegt je nach Quelle zwischen 0.004 kWh/GB und 136 kWh/GB [14]. Der zweite Faktor ist die CO2-Intensität des Stroms. Die CO2-Intensität gibt an, wie viel Gramm CO2 bei der Produktion einer Kilowattstunde Strom entsteht. Hier variieren die

[4]Kilowatt Stunden pro Gigabyte

Werte je nach Land und Energiequelle stark. Wenn man den Aufruf einer Website betrachtet, dann geht der Datenverkehr oft über mehrere Server und Netzwerke, die in unterschiedlichen Ländern stehen und unterschiedliche Energiequellen verwenden. Es ist also schwierig, die genaue CO2-Intensität zu bestimmen.
Festhalten lässt sich jedoch folgendes: Je geringer der Energieverbrauch und je sauberer der Strom, desto geringer sind die CO2-Emissionen. Deshalb ist es essenziell, die Effizienz der Website zu steigern und auf erneuerbare Energiequellen zu setzen.

1.2.4 Messung des Energieverbrauchs von Websites

Es gibt einige Tools, die mit der Metrik des *Page Weight* arbeiten. *Page Weight* gibt an, wie groß eine Website ist und wie viele Ressourcen sie lädt. Je größer die Website, desto mehr Daten müssen übertragen werden und desto mehr Energie wird verbraucht. Ein Tool, welches mittels *Page Weight* den Energieverbrauch von Websites schätzt, ist *Website Carbon*[5]. Hier kann man die URL einer Website eingeben und erhält eine Schätzung des Energieverbrauchs und der CO2-Emissionen pro Seitenaufruf. Auch dieses Tool arbeitet mit einer Schätzung und kann den tatsächlichen Energieverbrauch nicht genau bestimmen. Obwohl es weitaus mehr Faktoren, wie z.B. Herstellungskosten, Laufzeitkosten und Gewichtung von Netzwerk, Datenzentren und Endgeräten vornimmt, ist es lediglich eine Näherung [15]. Solange man Websites jedoch immer mit dem gleichen Tool misst, kann man sie vergleichen und Verbesserungen und Verschlechterungen feststellen. Es ist also generell empfehlenswert, das *Page Weight* von Websites gering zu halten.

1.2.5 Energieverbrauch auf Serverseite

Auf der Serverseite ist es wichtig, effizienten Code zu schreiben. Doch auch effizienter Code ist durch die Programmiersprache, in der er geschrieben ist, begrenzt. In einem Paper aus dem Jahr 2021 wird der Energieverbrauch von verschiedenen Programmiersprachen verglichen [16].

In der Tabelle 1.2[6] ist zu sehen, dass es große Unterschiede zwischen den Programmiersprachen gibt. Die Tabelle ist nach dem Energieverbrauch in Joule sortiert. Der Energieverbrauch wurde auf unterschiedlichen Benchmarks mit Intel's *Running Average Power Limit* gemessen [17]. Es zeigt sich, dass C die effizienteste Programmiersprache ist; gefolgt von Rust. Sie sind sowohl die Schnellsten in Bezug auf die Laufzeit in Millisekunden (ms) sowie den Energieverbrauch in Joule (J). Andere Programmiersprachen wie Python (75.88), JavaScript (4.45),

[5] https://www.websitecarbon.com

[6] anders dargestellt als im Original und nach dem Energieverbrauch sortiert, statt jede einzelne Spalte jeweils sortiert darzustellen

Tabelle 1.2: Programmiersprachen Benchmark für Energieverbrauch, Laufzeit, und Speicherbedarf [16]

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

Java (1.98) oder Rust (1.03) sind weniger effizient als C (1.00).[7] Aufgrund dieser Ergebnisse lässt sich der Energieverbrauch für die Serverseite allein durch die Wahl der Programmiersprache reduzieren. Auch, wenn der Energieverbrauch in der Praxis von vielen weiteren Faktoren abhängt, ist die Wahl der Programmiersprache ein wichtiger Faktor und guter Ausgangspunkt.

[7] in Klammern der Faktor um den sie mehr Energie verbrauchen als C

1.2.6 Energieverbrauch auf Netzwerkseite

Die Messung des Energieverbrauchs seitens des Netzwerks ist für diese Arbeit nicht relevant. Sie ist aus Sicht des Webentwicklers nur insofern zu beeinflussen, als dass weniger Datenverkehr gleich weniger Energieverbrauch bedeutet. Es kann jedoch keinen Einfluss auf die Technologien und Infrastruktur des Netzwerks genommen werden. Der Energieverbrauch des Netzwerks kann auch nur von denen gemessen werden, die die Infrastruktur betreiben.

Tabelle 1.3: Übersicht der Energieverteilung auf die verschiedenen Systemsegmente [15].

Systemsegment	**Energieanteil (%)**
Data centers	22%
Networks	24%
User devices	54%

Wie in der Tabelle 1.3 zu sehen ist, liegt der Energieverbrauch des Netzwerks bei 24% der verbrauchten Gesamtenergie. Es ist also ein wichtiger Aspekt im Gesamtbild des Energieverbrauchs von Websites.

1.2.7 Energieverbrauch aufseiten des Endgeräts

Der Energieverbrauch aufseiten des Endgeräts ist gegenwärtig gut messbar. Es gibt Tools, welche direkt in die Browser eingebaut sind und den Energieverbrauch von Websites auf Endgeräten messen können. Sowohl *Firefox* als auch *Safari* haben solche Tools eingebaut [18] [19]. Der *Firefox Profiler* gibt sehr detaillierte Einblicke in den Energieverbrauch der Website auf dem Endgerät. Es ist möglich, Zeitfenster zu wählen und den Energieverbrauch in diesem Zeitfenster zu messen. Außerdem ist es auf dem Endgerät wie auf dem Server möglich, den Energieverbrauch mittels *Running Average Power Limit* zu messen. In der Abbildung 1.1 ist ein Screenshot aus dem *Firefox Profiler* zu sehen. Hier wurde die für diese Arbeit erstellte Website auf dem *localhost* aufgerufen und der Energieverbrauch gemessen. Es ist zu sehen, wie der Energieverbrauch in Watt über die Zeit gemessen wird. In der Abbildung 1.2 ist ein Screenshot aus der Safari Timeline zu sehen. Auch hier wurde die gleiche Website wieder auf dem *localhost* aufgerufen und der Energieverbrauch gemessen. In Safari gibt es jedoch keine direkte Anzeige des Energieverbrauchs in Watt. Dafür ist hier übersichtlich dargestellt welche Prozesse auf der CPU laufen und wie viel Energie sie prozentual verbrauchen. Auch ohne genaue Angabe des Energieverbrauchs in Watt, ist es möglich Websites zu vergleichen. Um also möglichst wenig Energie zu verbrauchen, sollte man also auf Servern und Endgeräten möglichst energieeffizienten Code, in energieeffizienten Programmiersprachen, nutzen und über das Netzwerk möglichst wenig Daten senden. Die Messungen des Energieverbrauchs sind jedoch nur Schätzungen und können nicht exakt bestimmen wie viel Energie eine

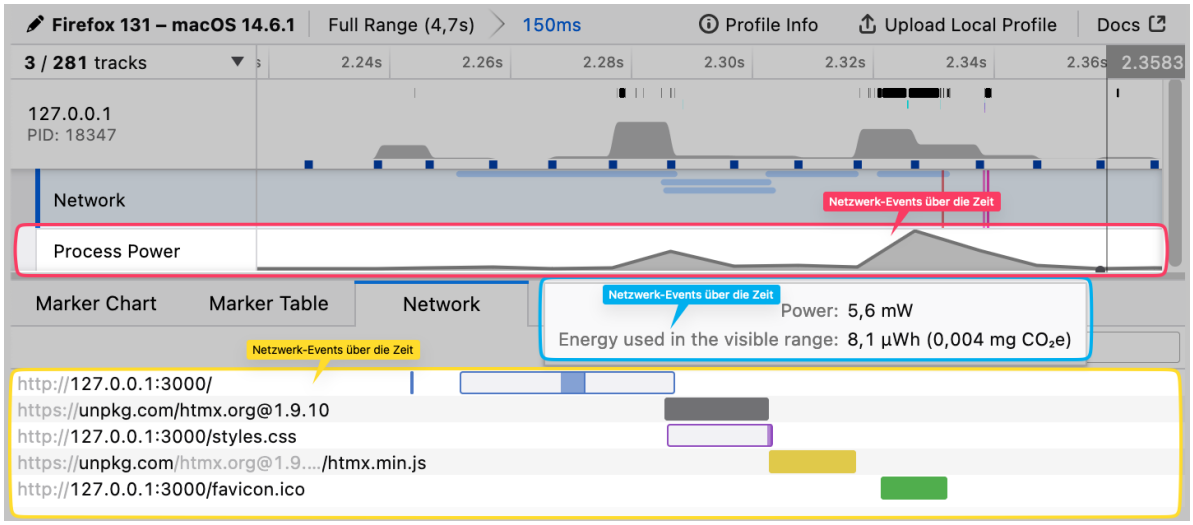

Abbildung 1.1: Screenshot aus dem *Firefox Profiler*

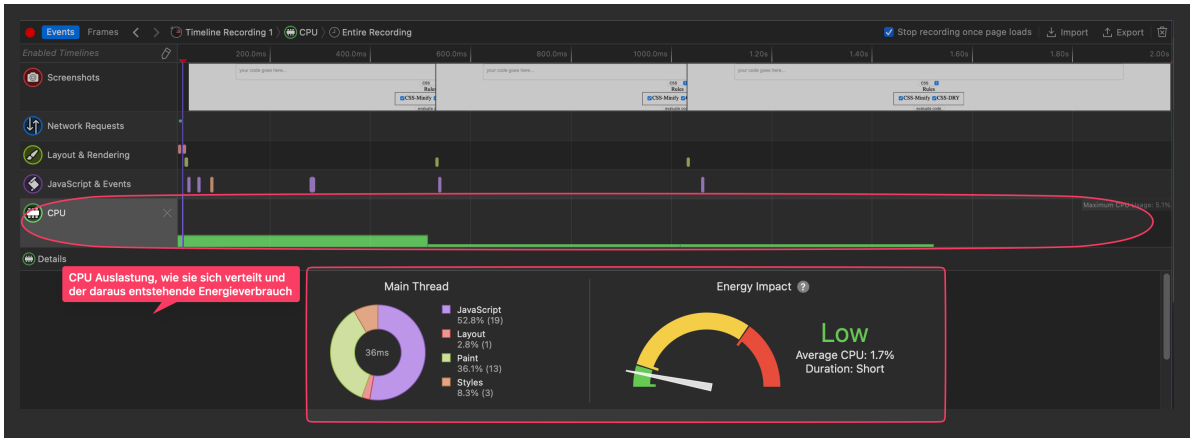

Abbildung 1.2: Screenshot aus der Safari Timeline

Website verbraucht. Aber sie können dennoch genutzt werden, um Websites zu vergleichen und Verbesserungen festzustellen. Hier sollte jedoch zur Messung immer das gleiche Tool verwendet werden, um die Ergebnisse vergleichbar zu machen.

1.3 Statische Programmanalyse

Für die Arbeit wurde ein Werkzeug entwickelt, welches mit statischer Programmanalyse Quellcode auf Energieeffizienz prüft. In diesem Kapitel soll es um die Grundlagen der statischen Programmanalyse gehen. Es wird erklärt, was statische Programmanalyse ist und wie sie funktioniert.

1.3.1 Definition

„Static program analysis aims to automatically answer questions about the possible behaviors of programs.“ [20]
Im vorangehenden Zitat wird der Zweck der statischen Programmanalyse beschrieben. Es geht darum, automatisiert Aussagen über das Verhalten von Programmen zu treffen. Diese Aussagen werden im Vorfeld getroffen, ohne das Programm auszuführen und basieren lediglich auf der Analyse des Quellcodes. Das unterscheidet statische Programmanalyse von dynamischer Programmanalyse, bei der das Programm ausgeführt wird, um Aussagen über das Verhalten zu treffen [21]. Beispiele für dynamische Programmanalyse sind *Unit Tests*, *Code Coverage*[8] oder *Performance* Analysen. Beides sind Programme, die Aussagen über andere Programme treffen.

1.3.2 Statische Programmanalyse und Compiler

Statische Programmanalyse ist ein Teil des Prozesses, der stattfindet, wenn aus Quellcode ein lauffähiges Programm entstehen soll. Der Prozess des Kompilierens besteht aus mehreren Schritten. Die Abbildung 1.3 zeigt die verschiedenen Wege, die der Quellcode nehmen kann,

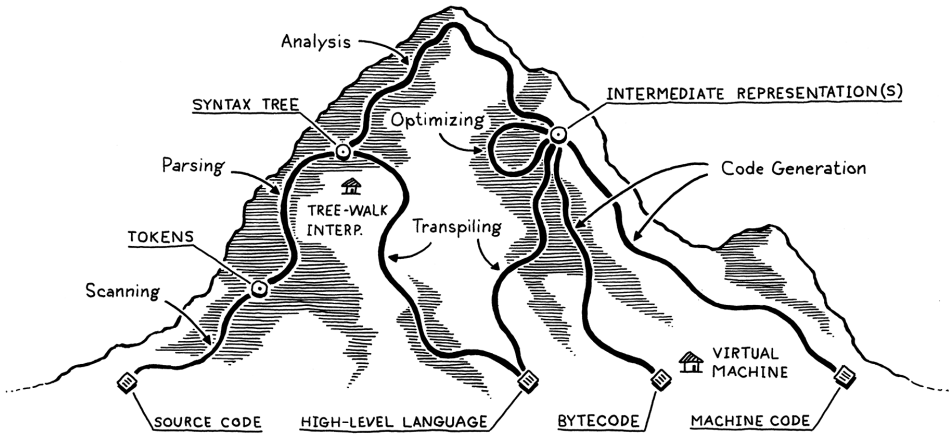

Abbildung 1.3: Die Wege des Quellcodes (Quelle: [23, Kapitel: A Map of the Territory])

wenn ein *Compiler* ihn bekommt. Am Fuße des Berges sind Quellcode und Maschinencode. Je höher man den Berg hinaufsteigt und die Schritte des *Compilers* durchläuft, desto mehr lässt sich aus dem Quellcode die Semantik ablesen. Am Gipfel des Berges wird versucht, die Semantik des Programms zu verstehen. Beim Absteigen des Berges wird der Maschinencode generiert. Die Semantik des Programms beim Erstellen des Maschinencodes berücksichtigt.

[8] beschreibt den Teil des Quellcodes, der beim Durchlaufen einer Anzahl von Tests erreicht wird [22]

So lassen sich Optimierungen vornehmen, die die Effizienz des Programms steigern, ohne dabei die Semantik zu verändern [23, Kapitel: A Map of the Territory].

1.3.3 Scanning

Der erste Schritt auf dem Weg vom Quellcode zum Maschinencode ist das *Scanning*. Hierbei

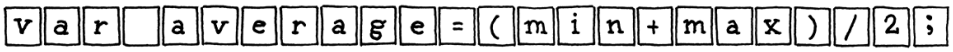

Abbildung 1.4: Input String [23, Kapitel: A Map of the Territory]

wird der Quellcode aus einer Anreihung von in Abbildung 1.4 in zusammengehörige Teile, die sogenannten *Tokens*, aufgeteilt. Die Unterteilung des gesamten Quellcodes in *Tokens* passiert mithilfe von Regeln, die in einer Grammatik festgelegt sind. Diese Grammatik nennt sich *lexikalische Grammatik* und beschreibt, wie der Quellcode in *Tokens* umgewandelt wird. Aus einem *String* aus Abbildung 1.4 werden *Tokens* in Abbildung 1.5. Hier werden oft auch

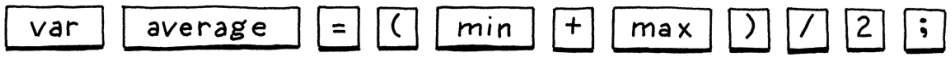

Abbildung 1.5: *Tokens* [23, Kapitel: A Map of the Territory]

Whitespaces und Kommentare entfernt, da diese für den Maschinencode keine Relevanz haben.

1.3.4 Parsing

Der nächste Schritt ist das *Parsing*. Aus den *Tokens* wird eine Baumstruktur gebildet, die den Regeln der Grammatik folgt. Diese Grammatik nennt sich *syntaktische Grammatik* und beschreibt, wie die *Tokens* wieder zu komplexeren Strukturen zusammengefasst werden. In der Regel sind diese Strukturen Bäume und heißen *Parse Trees* oder *Abstract Syntax Trees (AST)*.

Aus Zeichen werden *Tokens* und aus den *Tokens* werden *Expressions* siehe Tabelle 1.4.

Tabelle 1.4: Scanning und Parsing vgl. [23, Kapitel: Representing Code]

Terminologie	**Lexikalische Grammatik**	**Syntaktische Grammatik**
das Alphabet besteht aus:	Zeichen	Tokens
werden zusammengefasst zu:	Token	Expressions
implementiert durch den:	Scanner	Parser

Expressions sind größere Einheiten, die aus mehreren *Tokens* bestehen. Beide Schritte folgen Regeln, die in den jeweiligen Grammatiken festgelegt sind.

Codeblock 1.1: nicht von herkömmlichen Grammatiken zugelassener Quellcode [24]

```
1    clask C {int y/-&&^while var xyz += (((((((x;}
```

Grammatiken sind formale Sprachen und werden von Compilern implementiert. Sie bilden die grundlegende Syntax von Programmiersprachen und würden Quellcode wie in Codeblock 1.1 nicht kompilieren. So wäre es denkbar, dass der *Scanner* alle in Codeblock 1.1 verwendeten Zeichen kennt, der *Parser* aber nicht in der Lage ist, aus den *Tokens* einen *AST* zu erstellen. Der *Parser* würde einen Fehler werfen, da der Quellcode nicht den Regeln der Grammatik entspricht.

1.3.5 Abstract Syntax Trees

Der *AST* ist eine abstrakte Repräsentation des Quellcodes. Er ist eine Baumstruktur und hat eine andere Datenstruktur als der Quellcode, welche die Semantik des Programms abbildet. In Abbildung 1.6 ist zu sehen wie aus den *Tokens* ein *AST* wird, der aus einem *Statement* und mehreren *Expressions* besteht. Die Klammern, das Gleichheitszeichen und das Semikolon

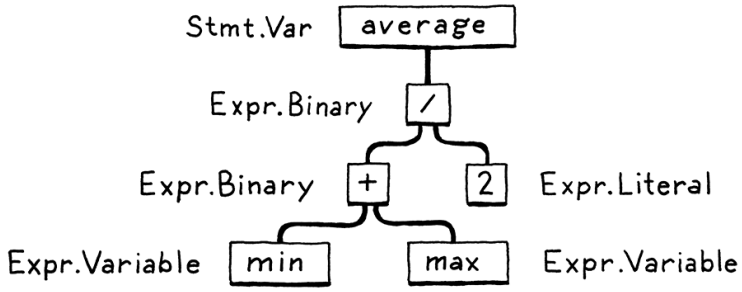

Abbildung 1.6: Abstract Syntax Tree [23, Kapitel: A Map of the Territory]

sind nicht mehr vorhanden. Die Struktur des Programms ist jedoch noch erkennbar ist. Außerdem haben die Knoten des Baumes alle einen *Typ*, der die Art des Knotens beschreibt. Es fallen syntaktische Details weg, die für den Maschinencode nicht relevant sind und die im *AST* anders repräsentiert werden [25]. Im Fall von JavaScript ist Grammatik durch die *ECMA-262* Spezifikation festgelegt [26]. Auch die für diese Arbeit verwendete *Library*, *JavaScript Oxidation Compiler (OXC)* implementiert diese Spezifikation [27]. Aufgrund dieser Spezifikation ist es möglich, den *AST* zu erstellen. Die Website *AST Explorer* [28] zeigt, wie der *AST* für ein gegebenes Programm aussieht.

Codeblock 1.2: JSON Darstellung eines *AST* für `var a` durch *AST Explorer*

```
1    {
2  "type": "Program",
3  "start": 0,
4  "end": 5,
5  "body": [
6    {
7      "type": "VariableDeclaration",
8      "start": 0,
9      "end": 5,
10     "declarations": [
11       {
12         "type": "VariableDeclarator",
13         "start": 4,
14         "end": 5,
15         "id": {
16           "type": "Identifier",
17           "start": 4,
18           "end": 5,
19           "name": "a"
20         },
21         "init": null
22       }
23     ],
24     "kind": "var"
25   }
26 ],
27 "sourceType": "script"
28 }
```

In Codeblock 1.2 ist der *AST* für den Codeschnipsel `var a` dargestellt. Der Einstieg ist immer ein `Program`-Knoten, der alle anderen Knoten enthält (Codeblock 1.2, Zeile 2). Darunter befindet sich ein `VariableDeclaration`-Knoten, der die Variable `a` deklariert (Codeblock 1.2, ab Zeile 7). Der `VariableDeclarator`-Knoten (Codeblock 1.2, ab Zeile 12) enthält die Information, dass die Variable den `Indentifier` `a` hat (Codeblock 1.2 Zeile 19) und nicht initiiert wurde (Codeblock 1.2, Zeile 21). So wird der *AST* für jedes Programm erstellt und kann dann weiterverarbeitet werden. Es gibt für alle Bestandteile eines Programms einen Knoten im *AST*. So kann das Programm in seiner Gesamtheit abgebildet werden [29].

1.3.6 Statische Analyse

Der *AST* ist die Grundlage für die statische Programmanalyse. Hier wird versucht, Aussagen über das Programm zu treffen, die auf der Semantik des Programms basieren.
„So, the parser says your program checks out according to the grammar. Doesn't mean it should be allowed to run, does it?" [24]
Das heißt, dass syntaktisch korrekter Quellcode nicht automatisch bedeutet, dass das Programm auch korrekt und lauffähig ist.

Codeblock 1.3: Syntaktisch eventuell korrekter Quellcode

```
1    class C {int y = x;}
```

Doch auch Codeblock 1.3 ist eventuell syntaktisch korrekt, aber sollte nicht zur Ausführung zugelassen werden. Hier wird statische Programmanalyse benötigt, um Aussagen über das Programm zu treffen, die über die Syntax hinausgehen. In statisch typisierten Sprachen wird hier z.B. geprüft, ob die Variablen korrekt initialisiert wurden, also mit Werten des richtigen Typs und bevor sie gelesen werden. Mit der statischen Programmanalyse können außerdem Optimierungen vorgenommen werden, die die Semantik des Programms nicht verändern. Häufiger vorkommende Berechnungen mit sich wiederholenden Termen, wie in Codeblock 1.4, sollten einmalig berechnet und als Konstante im Maschinencode gespeichert werden (siehe Codeblock 1.5).

Codeblock 1.4: Berechnung der Größe eines Geldstücks

```
1    pennyArea = 3.14159 * (0.75 / 2) * (0.75 / 2);
```

Codeblock 1.5: größe des Geldstücks als Konstante hinterlegt

```
1    pennyArea = 0.4417860938;
```

1.3.7 Anwendungen für statische Programmanalyse

Statische Programmanalyse unterteilt sich in drei Anwendungsfälle [20]:

1. **Programm-Optimierung:** Hier geht es um die Optimierung von Programmen beim Kompilieren, um effizienten Maschinencode zu generieren.

   - Hat ein Programm toten Code, der von der `main` nicht erreicht und beim Kompilieren entfernt werden kann? So könnte die Größe des Programms reduziert werden.
   - Ist das Ergebnis eines Ausdrucks innerhalb einer Schleife in jeder Iteration derselbe? Falls ja, kann der Ausdruck außerhalb der Schleife platziert werden, um überflüssige Berechnungen zu vermeiden.

   Dies sind Beispiele für Optimierungen, die durch statische Programmanalyse möglich sind.

2. **Fehlererkennung:** Hierbei wird versucht, Fehler im Programm zu finden. Es wird versucht zu beweisen, dass ein Programm eine bestimmte Eigenschaft nicht hat.

   - Gibt es *Null-Pointer-Dereferenzierungen*, die zu einem Programmabsturz führen könnten? Sollten welche gefunden werden, kann das Programm so angepasst werden, dass es nicht abstürzt.
   - Wurden Variablen initialisiert, bevor sie benutzt werden? Wenn nein, dann sollten diese initialisiert werden.

3. **Programmentwicklung:** Dieser Punkt bezieht sich auf die Entwicklung von Programmen. Hier werden Hinweise zum *Refactoring*, zur Verbesserung der Lesbarkeit und zum *Debugging* des Codes gegeben.

   - Welche Typen kann eine Variable annehmen? Ist klar eine Variable hat den falschen Typ, so kann darauf hingewiesen werden.
   - Gibt es duplizierten Code? Dann kann dieser ggf. in eine Funktion ausgelagert werden.

1.4 Das Visitor Pattern

Im folgenden Kapitel wird das *Visitor Pattern* vorgestellt. Es ist ein Entwurfsmuster, welches oft in Verbindung mit *Abstract Syntax Trees (AST)* verwendet wird. Da die Arbeit auf *AST's* arbeitet und die genutzte Bibliothek *OXC* das *Visitor Pattern* implementiert, ist es sinnvoll, das *Visitor Pattern* zu verstehen. So kann die angebotene Schnittstelle des *OXC Crates* genutzt werden.
Die folgenden Abschnitte beschäftigen sich mit dem *Visitor Pattern* und basieren auf dem Buch *Design Patterns: Elements of Reusable Object-Oriented Software* von Erich Gamma, Richard Helm, Ralph Johnson und John Vlissides [30]. Das Buch ist auch als *Gang-of-Four*-Buch bekannt und ist ein Standardwerk für Entwurfsmuster.

1.4.1 Zweck des Visitor Patterns

Das Visitor Pattern ist ein Entwurfsmuster, das es ermöglicht Operationen auf einer Menge von unregelmäßigen Objekten auszuführen[9], ohne die Klassen dieser Elemente zu verändern. Die betroffenen Klassen sind typischerweise in einer Klassenhierarchie organisiert und es muss lediglich im Vorfeld vorgesehen werden, dass die Elemente Visitor akzeptieren können.

1.4.2 Motivation für die Verwendung des Visitor Patterns

Ein Bestandteil dieser Arbeit ist der Umgang mit *Abstract Syntax Trees (AST)*. Diese Art von Datenstruktur ist unregelmäßig, da sie aus verschiedenen Typen von Knoten besteht, die unterschiedliche Daten enthalten. Außerdem gibt es viele unterschiedliche Operationen, die auf einem *AST* ausgeführt werden sollen. Unter anderem sollen Operationen wie z.B. *Pretty Printing*[10], *Typ-Überprüfung* und *Code-Optimierungen* auf einem *AST* ausgeführt werden. Die *ASTs* haben für die unterschiedlichen *Node*-Typen unterschiedliche Implementierungen, welche innerhalb einer Klassenhierarchie organisiert sind. Die expliziten Implementierungen der Klassenhierarchie unterscheiden sich von Programmiersprache zu Programmiersprache, sind jedoch in der Regel in einer Klassenhierarchie organisiert.
Auf Abbildung 1.7 ist ein Teil der Klassenhierarchie eines *ASTs* dargestellt. Hier werden innerhalb der Klassen jeweils die Methoden implementiert. Dies führt zu einer unübersichtlichen und schwer wartbaren Struktur. Die entsprechenden Klassen würden sehr groß werden, neue Funktionalitäten müssten in jeder Klasse implementiert werden und es wäre schwierig, die

[9] im Gegensatz zu einem Iterator, der es ermöglicht Operationen auf einer Menge gleichartiger Elemente (Collections) auszuführen
[10] Optimierung der Codeausgabe zur besseren Lesbarkeit

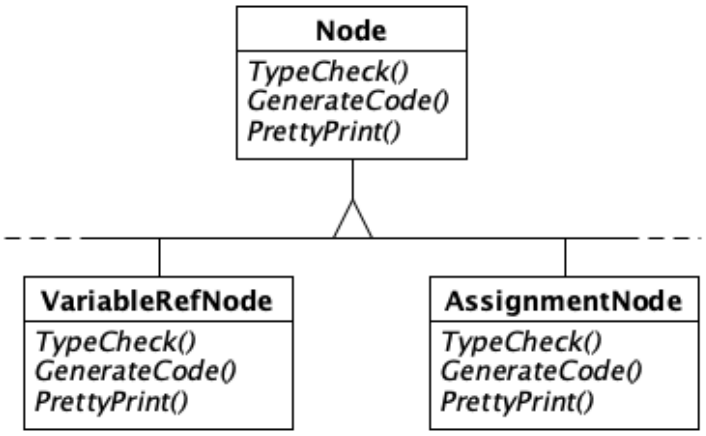

Abbildung 1.7: Ein Teil der *Node*-Klassenhierarchie [30, S. 401]

Klassen zu erweitern oder zu verändern. Wird neues Verhalten hinzugefügt, müssten außerdem alle Klassen neu kompiliert werden. Es wäre wünschenswert, wenn es eine Möglichkeit gäbe die Klassen zu erweitern, ohne die Klassen selbst zu verändern. Um diese Probleme zu lösen, können Funktionen in Klassen gekapselt werden, die als *Visitor* bezeichnet werden. Das *Visitor*-Objekt wird beim Traversieren des *ASTs* an die *Nodes* übergeben. *Nodes* 'akzeptieren' den *Visitor* und rufen die, für die eigene Klasse zuständige, Methode auf dem *Visitor* auf. Also führt nun der *Visitor* die Operationen auf den *Nodes* aus und nicht mehr die *Nodes* selber. Die Klassen müssen nicht mehr angepasst werden. Statt der `PrettyPrint()`-Methode in jeder Klasse gibt es nun ein `PrettyPrintVisitor`-Objekt, welches erzeugt wird und dem *AST* mittels `accept` übergeben wird. Jede *Node* implementiert die `accept`-Methode und ruft die entsprechende `visit`-Methode auf dem *Visitor* auf. So wird aus `PrettyPrint`-Methode in der Klasse `VariableRefNode` nun die `VisitVariableRef`-Methode im `PrettyPrintVisitor`-Objekt. Damit neue Funktionalitäten hinzugefügt werden können, muss nun noch ein allgemeines Interface für die *Visitor* implementiert werden, welches die Methoden für die verschiedenen *Nodes* enthält. Das Interface `NodeVisitor` enthält also die Methoden `VisitVariableRef` und `VisitAssignment`. Der anwendungsspezifische Code verschwindet aus den Klassen und wird in den konkreten Unterklassen des `NodeVisitor` Interfaces implementiert. Das *Visitor Pattern* ermöglicht die Kapselung der Operationen in *Visitor*-Objekten und die Erweiterung der Funktionalität, ohne die Klassen zu verändern.

Das Design Pattern *Visitor* definiert also zwei Klassenhierarchien (Abbildung 1.8, 1.9). Eine für die *Nodes*, die bearbeitet bzw. gelesen werden. Die andere für die *Visitor*, die die entsprechenden Operationen zum Bearbeiten und Lesen definieren.

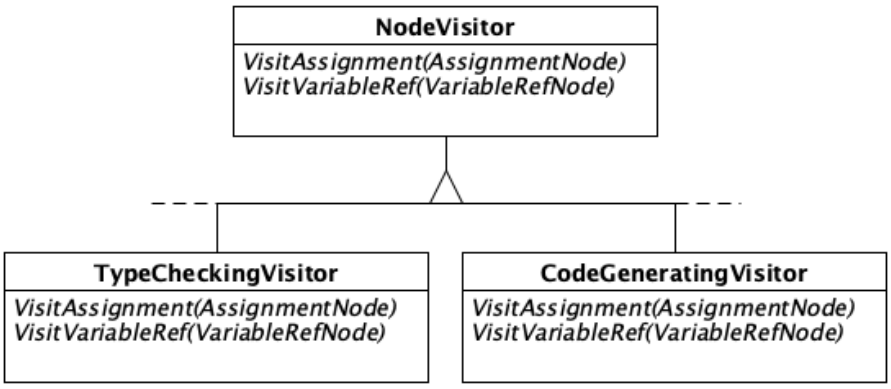

Abbildung 1.8: Die `NodeVisitor`-Hierachie [30, S. 402]

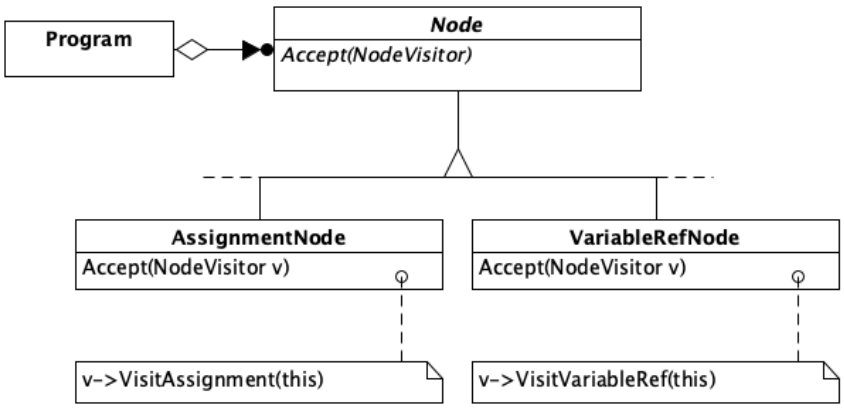

Abbildung 1.9: Die Node-Hierachie [30, S. 403]

1.4.3 Anwendbarkeit des Visitor Patterns

Das *Visitor Pattern* ist sinnvoll anzuwenden, wenn:

- es eine Objektstruktur vieler unterschiedlicher Klassen gibt, die unterschiedliche Schnittstellen haben. Zugleich ist es sinnvoll anzuwenden wenn, die Operationen, die auf den Objekten ausgeführt werden sollen, von den konkreten Klassen abhängen.
- auf der Objektstruktur viele unterschiedliche Operationen ausgeführt werden sollen, ohne die Klassen der Objekte zu verändern und diese nicht zu groß werden zu lassen. Durch die Kapselung der Operationen in *Visitor*-Objekten ist es möglich, unterschiedlichen Anwendungen nur bestimmte Teile zur Verfügung zu stellen.
- die Objektstruktur sich nur selten ändert, jedoch oft neue Operationen hinzugefügt werden sollen. Für jede Änderung an der Objektstruktur müssten alle *Visitor*-Klassen angepasst werden. Gibt es also häufige Änderungen an der Objektstruktur, ist das

Visitor Pattern nicht geeignet und es sollte in Betracht gezogen werden, die Klassen selbst zu ändern.

1.4.4 Struktur des Visitor Patterns und Teilnehmer

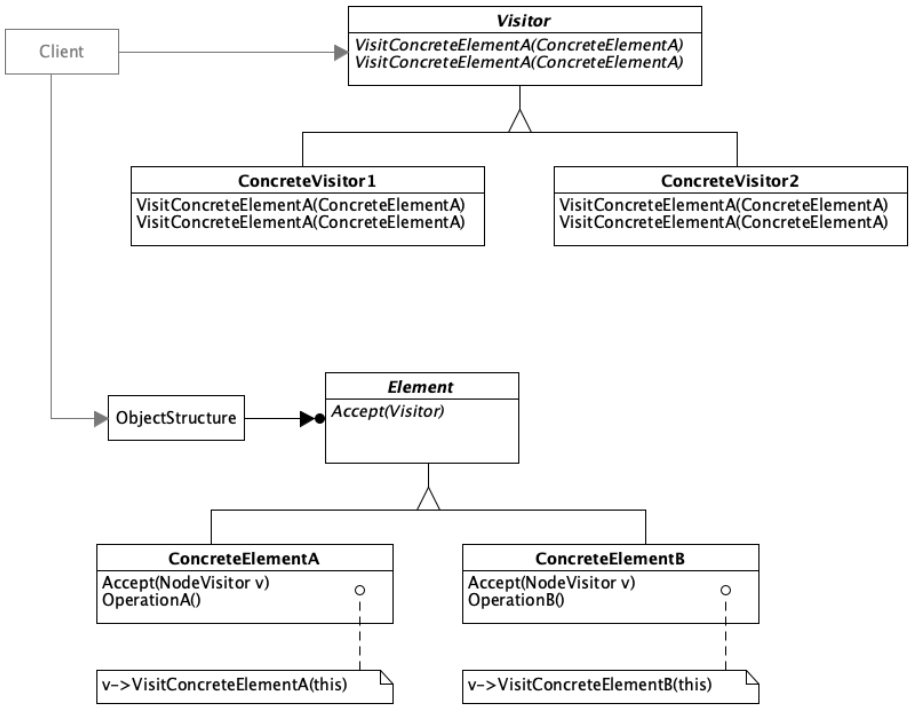

Abbildung 1.10: Struktur des *Visitor* Patterns [30, S. 404]

Abbildung 1.10 zeigt die Struktur des *Visitor Patterns* mit den folgenden Teilnehmern.

- **Visitor**: Definiert eine Methode für jeden konkreten Elementtyp `ConrecteElement` in der Objektstruktur. Name und Signatur der Methode geben an, welcher `ConrecteElement`-Typ besucht (`VisitConcreteElement(this)`) wird. So ist immer klar welches `ConcreteElement` besucht wird und kann direkt auf die Schnittstelle des `ConcreteElement` zugreifen.
- **ConcreteVisitor**: Implementiert die Methoden, die von `Visitor` definiert wurden. So können unterschiedliche Algorithmen durch unterschiedliche `ConcreteVisitor` implementiert werden.
- **Element**: Definiert eine Methode `Accept` mit einem `Visitor` als Argument.
- **ConcreteElement**: Implementiert die `Accept`-Methode und ruft die entsprechende Methode des `Visitor` mit sich selbst als Argument auf.

- **ObjectStructure**: Kann eine Sammlung von `Element`-Objekten sein und diese der Reihe nach benennen. Sie kann gegebenenfalls eine High-Level-Schnittstelle zum Traversieren der Elemente für den `Visitor` bereitstellen und ist in der Regel entweder vom Typ *Composite* oder *Collection*.

1.4.5 Sequenzdiagramm des Visitor Patterns

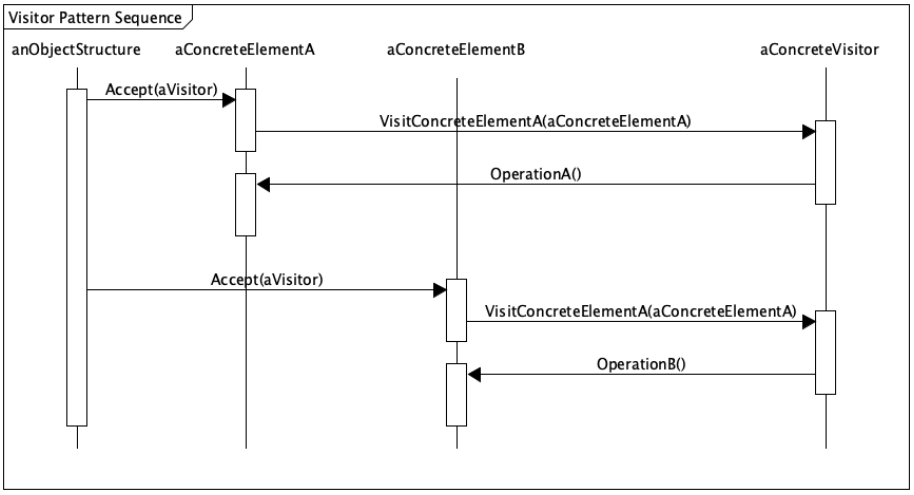

Abbildung 1.11: Sequenzdiagramm mit *Objektstruktur*, *Visitor*-Objekt und zwei konkreten *Elementen* [30, S. 405]

Abbildung 1.11 zeigt ein Sequenzdiagramm des *Visitor Patterns*. Ein *Client* kann beim Traversieren der *Objektstruktur* die konkreten *Elemente* mit einem `Visitor`-Objekt besuchen. Beim Besuch wird die `Accept`-Methode des *Elements* und die entsprechende `Visit`-Methode des `Visitor`-Objekts aufgerufen. Das `Visitor`-Objekt kann dann auf die Schnittstelle des *Elements* zugreifen und die Operationen `OperationA()` und `OperationB()` ausführen.

1.4.6 Fazit zum Visitor Pattern

1. Das Hinzufügen neuer Operationen wird durch das *Visitor Pattern* vereinfacht, da für neue Operationen nur ein neues `Visitor`-Objekt erstellt werden muss. Die Klassen der gegebenenfalls komplexen *Objektstruktur* müssen nicht verändert werden, da die Operationen in den `Visitor`-Objekten gekapselt sind.
2. Des Weiteren ist es möglich, ähnliche Operationen in einem `Visitor`-Objekt zu kapseln und unterschiedliche voneinander zu trennen. So werden sowohl die Klassen der *Objektstruktur* als auch die `Visitor`-Objekte entkoppelt und vereinfacht.

3. Das Hinzufügen neuer `ConcreteElement`-Klassen ist jedoch schwieriger, da für jede neue `ConcreteElement`-Klasse eine neue `Visit`-Methode in jedem `Visitor`-Objekt erstellt werden muss. Es können zwar Standardimplementierungen für die `Visit`-Methoden erstellt werden, dies ist jedoch nicht immer ausreichend, da die Operationen auch von den konkreten neuen `ConcreteElement`-Klassen abhängig sind. Daraus ergeben sich die zwei Optionen, ob sich die auf eine *Objektstruktur* angewendeten Algorithmen häufiger ändern bzw. neue hinzukommen oder ob sich die Struktur der Klassen häufiger ändert. Die Klassenhierarchie der `Visitor`-Klassen ist schwierig zu warten, wenn häufig neue `ConcreteElement`-Klassen hinzugefügt werden. Hier wäre es sinnvoller, die Klassen der *Objektstruktur* direkt für die Operationen zu erweitern. Gibt es eine feste *Objektstruktur*, welche sich nicht verändert, aber häufig neue Algorithmen hinzugefügt oder verändert werden, ist das *Visitor Pattern* hilfreich.

4. Ein weiterer Faktor, der berücksichtigt werden sollte, ist die Beschaffenheit der *Objektstruktur* in Bezug auf die Elementtypen, die besucht werden. Hier kann zwischen gleichartigen und unterschiedlichen Elementtypen unterschieden werden. Während bei gleichartigen Elementtypen ein *Iterator*-Pattern anwendbar ist, ist dies bei unterschiedlichen Elementtypen nicht möglich. [11] Hier ist das *Visitor Pattern* sinnvoll, da es Operationen auf unterschiedlichen Elementtypen ausführen kann. Es bietet mit der `Visitor`-Schnittstelle die Möglichkeit, Objekte mit beliebigen Typen zu besuchen (`Accept`-Methode) und Operationen auf diesen auszuführen (`Visit`-Methode).

5. Was auch berücksichtigt werden sollte, ist, dass die `Visitor`-Objekte einen Zustand haben können, der den Besuch der unterschiedlichen Elemente überlebt. So können die `Visitor`-Objekte Informationen sammeln, die von den besuchten Elementen abhängig sind und diese Informationen für spätere Operationen nutzen. Es ist nicht notwendig, eine weitere Schnittstelle zu implementieren, um Informationen zwischen den Operationen auszutauschen.

6. Für die `Visitor`-Objekte ist es eine Schnittstelle zu den `ConcreteElement`-Objekten notwendig. Dies hat zur Folge, dass die `ConcreteElement`-Objekte eine öffentlich zugängliche und ausreichende Schnittstelle zur Verfügung stellen müssen. Dadurch wird die Kapselung der `ConcreteElement`-Objekte verletzt, da die `Visitor`-Objekte auf die Schnittstelle der `ConcreteElement`-Objekte zugreifen können. Ist die unerwünscht, so ist das *Visitor Pattern* nicht geeignet.

[11] Das *Iterator*-Pattern ermöglicht es, Operationen auf einer Menge gleichartiger Elemente auszuführen. Es wird in der Regel auf *Collections* angewendet, die eine Menge gleichartiger Elemente enthalten. Die Grundvoraussetzung ist, dass die Elemente der *Collection* alle den gleichen Typ haben.

1.4.7 Double Dispatch

Das Visitor Pattern simuliert *„Doppelte Verteilung“ engl. Double Dispatch* [12], um die Methode, die aufgerufen werden soll, von den Typen zweier Klassen abhängig zu machen. In Programmiersprachen, die nur *Single Dispatch* unterstützen, sind zwei Methodenaufrufe notwendig, um die aufzurufende Methode, von den Typen zweier Klassen abhängig zu machen. *Single Dispatch* bedeutet, dass die Methode, die aufgerufen wird, vom Typ des Empfängerobjekts und vom Namen des Requests abhängig ist. *Double Dispatch* bedeutet, dass die Methode, die aufgerufen wird, vom Typ zweier Empfängerobjekte und vom Namen des Requests abhängig ist. Die `Accept`-Operation[13] ist hierbei der *Double Dispatch.* Dies bedeutet, dass die Operation von dem Typ des *Visitors* und vom Typ des *Elements* abhängig ist. Der erste *Dispatch* passiert bei der *dynamischen Bindung*[14] der Methode `Accept` durch den Typ des `Elements`, auf dem sie aufgerufen wird. Dies ist möglich, da die Methode `Accept` in dem *Element* Interface deklariert ist und somit in den konkreten Klassen `ConcreteElement` überschrieben werden muss. Der zweite *Dispatch* passiert, wenn auf dem vorher übergebenen, konkreten Visitor die Methode `Visit` aufgerufen wird. Dieser Methodenaufruf findet in der konkreten Klasse des Elements `ConcreteElement` statt und als `Element` wird `this` übergeben. An diesem Punkt ist die aufzurufende Methode eindeutig, da sie von der konkreten Klasse des Elements `ConcreteElement` abhängig ist und innerhalb dieser Klasse aufgerufen wird (`this`). Zwei Methodenaufrufe sind notwendig, um die Methode, die aufgerufen werden soll, von den Typen zweier Klassen abhängig zu machen. Dies ist in Programmiersprachen, die *Multiple Dispatch* unterstützen, nicht notwendig, da sie die Methode, die aufgerufen werden soll, von den Typen zweier oder mehrerer Klassen abhängig machen können. Deshalb ist es in Programmiersprachen, die keinen *Multiple Dispatch* unterstützen, ratsam, das Pattern *Visitor* zu verwenden und in Programmiersprachen, die *Multiple Dispatch* unterstützen, unnötig. Das Schlüsselkonzept des *Visitor Patterns* ist es, die Operationen von konkreten Klassen und den Besuchern abhängig zu machen. Die Operationen werden nicht in den konkreten Klassen implementiert, sondern in den *Visitor*-Objekten, die die konkreten Klassen besuchen.

[12] *Double Dispatch* ist ein Fall von *multiple-dispatch dt. Mehrfache Verteilung* , mit *Double Dispatch* ist es möglich einen Methodenaufruf von mehreren Klassen abhängig zu machen.

[13] Operation bezeichnet hier nicht einen konkreten Methodenaufruf, sondern eine Prozedur, die in diesem Fall zwei Methodenaufrufe beinhaltet.

[14] Ermitteln des Typs eines Objekts zur Laufzeit statt beim Kompilieren (statische Bindung)

1.5 Rust

Für diese Arbeit wurde die Programmiersprache Rust verwendet. Folgend werden die Besonderheiten und Vorteile von Rust erläutert. Es soll gezeigt werden, wie Rust *Memory Safety* und *Performance* vereint.

1.5.1 Warum Rust?

Wie im Kapitel 1.2 bereits beschrieben, ist Rust nach C die energieeffizienteste und schnellste Programmiersprache (siehe Tabelle 1.2). Rust ist eine moderne Programmiersprache, die von Mozilla entwickelt wurde. Sie hat in den letzten Jahren immer mehr an Popularität gewonnen. Laut des *Stack Overflow Developer Survey 2023* ist Rust seit acht Jahren die *Most Desired*, vorher *Most Loved*, Programmiersprache. Das bedeutet, dass Nutzer nach dem Gebrauch von Rust am häufigsten angegeben haben, dass sie die Programmiersprache wieder verwenden würden [31]. Rust ist eine Programmiersprache, die sich durch ihre Robustheit und Geschwindigkeit auszeichnet. Auf der offiziellen Website wird Rust als „A language empowering everyone to build reliable and efficient software“ beschrieben [32]. Besonderheiten sind, die Performance, Zuverlässigkeit und Produktivität. Performance wird durch den Verzicht auf *Garbage Collection* erreicht. Zuverlässigkeit wird durch das *Ownership Model*, welches *Memory* und *Thread Safety* garantiert, erreicht. Produktivität schaffen eingebaute Tools, wie *Cargo* (Paketmanager [33]), *Rustfmt* (Codeformatierung [34]), *Clippy* (Linter [35]) und hilfreiche Fehlermeldungen des *Compilers*. Für diese Arbeit wurde Rust hauptsächlich wegen der Energieeffizienz gewählt.

1.5.2 Das Ownership Model in Rust

Rust hat einige Besonderheiten, die es von anderen Programmiersprachen unterscheidet. Das *Ownership Model*, ist das Feature, mit der größten Besonderheit und hat weitreichende Implikationen [36]. Dieses Feature stellt *Memory Safety* sicher, ohne dazu einen *Garbage Collector* zu benötigen. *Memory Safety* bedeutet, dass ein Programm nie auf Speicher zugreift, der außerhalb des, für das Programm reservierten, Speicherbereichs liegt. Außerdem bedeutet es, dass nie Instruktionen ausführt werden, die außerhalb des Codebereichs liegen, welcher in dem vorher genannten Speicherbereich angelegt wurde [37].
Alle Programme müssen Speicher verwalten. Dies geschieht in den meisten Programmiersprachen durch *Garbage Collection* oder selbstständiges *Allokieren* und *deallokieren* von Speicher. *Garbage Collection* dient als automatischer Speicher-Manager, da manuelle Speicherverwaltung fehleranfällig sein kann. Der *Garbage Collector* verwaltet die Belegung und Freigabe von Arbeitsspeicher für eine Anwendung. Die Speicherverwaltung muss also nicht mehr manuell gemacht werden. Dadurch kann es nicht mehr passieren, dass nicht mehr benötigter Speicher

nicht freigegeben wird. Dies würde zu Arbeitsspeicher-Verlusten führen. Außerdem wird vermieden, dass auf Speicher zugegriffen wird, der bereits für ein anderes Objekt freigeben wurde [38]. *Garbage Collection* hat jedoch auch Nachteile, da es an ständiger Überwachung des Speichers bedarf. Dies kann zu Verlusten bei der Performance durch den zusätzlichen *Overhead* führen. Außerdem können unvorhersehbare *Lags* entstehen, wenn der *Garbage Collector* den Speicher aufräumt.
Rust wählt einen dritten Weg, der *Memory Safety* mit der Performance von manueller Speicherverwaltung kombiniert, das *Ownership Model*. Das *Ownership Model* sind eine Menge von Regeln, die durch den *Compiler* überprüft werden können und bei deren Verletzung das Programm nicht kompiliert. Dieses Konzept führt nicht zu Performance-Verlusten, stellt aber *Memory Safety* sicher. Die Regeln sind [36]:

1. Jeder Wert in Rust hat einen *Owner*.
2. Es kann nur einen *Owner* zurzeit geben.
3. Wenn der *Owner* das *Scope*[15] verlässt, wird der Wert gelöscht und der Speicher freigeben.

Stack und Heap

Eine wichtige Rolle spielt hier die Unterscheidung zwischen *Stack* und *Heap*.
Der *Stack* ist ein Stapel, auf dem Werte gespeichert werden, die zur Kompilierzeit eine bekannte Größe haben. Dies sind in der Regel primitive Datentypen wie z.B. *Integer*, *Boolean*, *Floats* oder auch *String Slices*[16]. Der *Stack* ist schnell. Hier werden Werte mit *Push* und *Pop* auf den Stapel gelegt und wieder entfernt.
Der *Heap* ist langsamer. Hier werden Datenobjekte gespeichert, die zur Kompilierzeit eine unbekannte Größe haben und diese zur Laufzeit verändern können. Hier werden komplexe Datenstrukturen wie *Strings*[17] oder *Vektoren* gelagert. Außerdem gibt es hier keine konkrete Ordnung, wie auf dem *Stack*. Für Daten, die auf dem *Heap* abgelegt werden sollen, muss Speicher allokiert und deallokiert werden. Deshalb brauchen alle Daten, die auf dem *Heap* abgelegt werden, einen *Pointer* zu dem Speicherbereich, der auf dem *Heap* allokiert wurde. Dieser *Pointer* wird auf dem *Stack* mittels *Push* abgelegt. So kann zur Laufzeit auf die Datenobjekte auf dem *Heap* zugegriffen werden.
Diese Unterscheidung ist wichtig für das *Ownership Model*, da es sich um die Verwaltung der Daten auf dem *Heap* kümmert. Wenn Datenobjekte das *Scope* verlassen, müssen sie aus

[15] Scope ist der Bereich in dem ein Wert gültig ist
[16] *String Slices* sind ein besonderer *String* Typ in Rust. Sie beschreiben Zeichenketten einer festen Länge, sind nicht veränderbar und können deshalb auf dem *Stack* gelagert werden
[17] *Strings* sind in Rust veränderbare Zeichenketten die zur Laufzeit ihre Größe verändern können und deshalb auf dem *Heap* gelagert werden müssen.

dem Speicher gelöscht werden. Dazu wird die `drop`-Methode automatisch auf der Variable, die das Datenobjekt hält, aufgerufen und so der Speicherbereich auf dem *Heap* freigegeben. Damit dies funktioniert, muss sichergestellt werden, dass es nur einen *Owner* pro Wert oder Datenobjekt gibt. Speicherbereiche sollten nie mehrmals freigegeben werden da, dies zu fehlerhaftem Verhalten führen kann.

Copy und Move

Codeblock 1.6: Ownership-Copy in Rust

```
    fn main() {
        let x = 5;
        let y = x; // x wird kopiert nach y
        println!("x = {}, y = {}", x, y); // x = 5, y = 5
    }
```

Auf dem *Stack* liegende Werte werden kopiert, wenn sie einer anderen Variable zugewiesen werden. Dies passiert in Codeblock 1.6. So sind beide Variablen gültig und zeigen auf unterschiedliche Werte auf dem Stack. Beim Verlassen des *Scopes* werden beide Werte gelöscht. In den meisten Programmiersprachen gilt ähnliches auch für Datenobjekte auf dem *Heap*. Dort wird der *Pointer* auf den *Heap* kopiert, beide Variablen sind gültig und zeigen auf die gleichen Datenobjekte auf dem *Heap*.
In Rust wird das Datenobjekt jedoch nicht kopiert, sondern bewegt (siehe 1.7), wenn es sich um Datenobjekte handelt, die auf dem *Heap* liegen. Dies führt im Anschluss dazu, dass die Variable, aus der das Datenobjekt bewegt wurde, nicht mehr gültig ist.

Codeblock 1.7: Ownership-Move in Rust

```
fn main() {
    let s1 = String::from("hello");
    let s2 = s1; // s1 move nach s2
    println!("{}", s1); // Fehler, s1 nicht mehr gültig wegen move
}
```

Hier die drei möglichen Szenarien in Codeblock 1.7:

1. Der *Pointer* aus Variable `s1` wird nach `s2` kopiert und beide Variablen sind gültig. Siehe Abbildung 1.12. Dies ist das häufigste Szenario in den meisten Programmiersprachen.

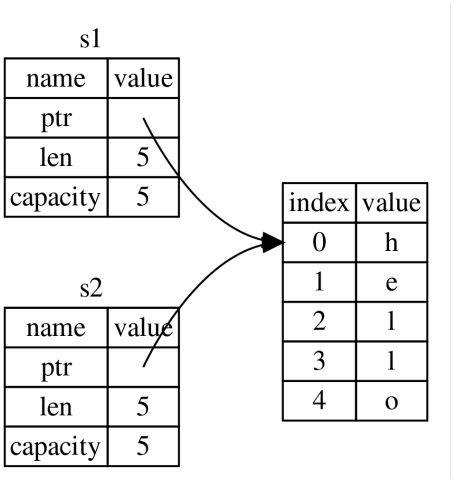

Abbildung 1.12: Kopie des *Pointers* auf den *Heap* [36]

2. Die *Heap* Daten von `s1` werden nach kopiert `s2` und ein neuer *Pointer* für `s2` angelegt. Beide Variablen, sind gültig und zeigen auf unterschiedliche Datenobjekte, siehe Abbildung 1.13. Dieser Vorgang ist aufwendig und wird deshalb in der Regel nicht gemacht, kann aber manuell mittels `clone`-Methode getätigt werden.[18]

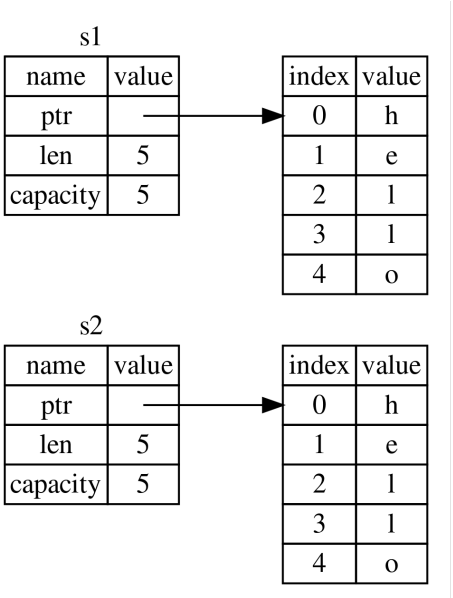

Abbildung 1.13: Vollständige Kopie [36]

3. Das Datenobjekt wird von `s1` nach `s2` bewegt und `s1` ist nicht mehr gültig, siehe Abbildung 1.14. Dieser Vorgang durchläuft in Rust.

[18]Dies ist dann eine tiefe Kopie, bei der auch die Daten auf dem *Heap* kopiert werden.

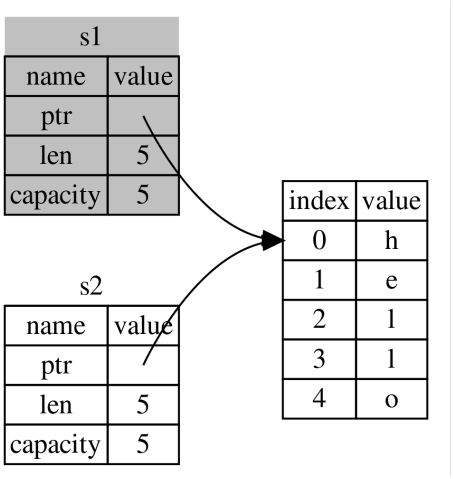

Abbildung 1.14: Bewegung der Referenz (Quelle: [36])

In Codeblock 1.7 wird der *String* `s1` nach `s2` mit `move` bewegt (Abbildung 1.14). Das bedeutet, dass `s1` nicht mehr gültig ist und nicht mehr verwendet werden kann. Dies ist ein wichtiger Punkt, der Rust von anderen Programmiersprachen unterscheidet. In anderen Programmiersprachen wird häufig eine Kopie des *Pointers* auf das Datenobjekt erstellt, wie in Abbildung 1.12 zu sehen. Danach werden vom *Garbage Collector* alle Referenzen aufgeräumt, die nicht mehr verwendet werden. Würde Rust auch die Gültigkeit beider Referenzen zulassen, dann würde die `drop`-Methode zweimal Speicher freigeben. Sie würde auf beiden Variablen aufgerufen werden, was zu einem *Double-Free*[19]-Fehler führen würde [36].
Es wird sichergestellt, dass immer nur ein *Owner* auf einen Wert oder ein Datenobjekt zeigt und dieser *Owner* für die Freigabe des Speichers verantwortlich ist, den Speicher freizugeben, wenn er das *Scope* verlässt. So schafft Rust es, *Memory Safety* (durch einen *Owner* zur Zeit) zu gewährleisten, ohne auf *Garbage Collection* zurückgreifen zu müssen (durch automatisiertes aufrufen von `drop`). Dies macht Rust zu einer der sichersten und performantesten Programmiersprachen, die es gibt. Rust wurde für diese Arbeit aufgrund der Energieeffizienz gewählt. Doch auch die Sicherheit ist nicht zu vernachlässigen. Laut eines Mitarbeiters bei Microsoft, waren beispielsweise 70% aller Sicherheits-Bugs auf *Memory-Safety*-Probleme zurückzuführen [40].

[19] „The product calls free() twice on the same memory address, potentially leading to modification of unexpected memory locations.“ [39]

1.5.3 Begrifflichkeiten in Rust

Crates und Cargo

Crates sind Einheiten in Rust, in denen Code organisiert wird. Diese können *Libraries* oder *Projekte* sein. Sie sind vergleichbar mit *Packages / Libraries* in anderen Programmiersprachen. Der *Package-Manager* wird direkt mit Rust mitgeliefert und nennt sich *Cargo*. Außerdem ist *Cargo* ein Build-System, das die Kompilierung und das Testen von Rust-Projekten automatisiert. *Cargo* wird auch dafür verwendet, *Crates* zu veröffentlichen [33]. Diese sind dann auf verfügbar und können in anderen Projekten verwendet werden [41].

Structs

Structs sind mit Klassen in Java oder anderen Programmiersprachen vergleichbar. Es gibt *Structs* mit Feldern, die Namen haben, *Tuple-Structs* mit Feldern ohne Namen und *Unit-Structs*, die keine Felder haben. Diese sind in Codeblock 1.8 zu sehen. Alle *Structs* können Methoden enthalten, die auf den *Structs* aufgerufen werden können.

Codeblock 1.8: *Structs* in Rust

```
1    struct Regular {
2        field1: f32,
3        field2: String,
4        pub field3: bool
5    }
6
7    struct Tuple(u32, String);
8
9    struct Unit;
```

Reguläre *Structs* werden am häufigsten verwendet. Sie sind wie Klassen in anderen Programmiersprachen. *Tuple-Structs* werden verwendet, wenn der Name des Tupels schon auf den Inhalt hinweist und die Felder keinen Namen brauchen (z.B. ein *Punkt Struct* mit zwei Koordinaten). *Unit-Structs* bieten die Möglichkeit, ein *Struct* zu erstellen, das keine Daten enthalten muss, aber ein *Trait* implementieren soll.

Traits

Traits sind vergleichbar mit Interfaces anderer Programmiersprachen und definieren eine Menge von Methoden. Sie können Standardimplementierungen für Methoden enthalten, die

dann von den implementierenden *Structs* überschrieben werden können. Außerdem können *Traits* genutzt werden, um zu erzwingen, dass Methoden von den implementierenden *Structs* implementiert werden müssen. *Structs* können mehrere *Traits* implementieren und diese Implementierungen auch für Übersichtlichkeit voneinander trennen, wie in Codeblock 1.9 zu sehen ist.

Codeblock 1.9: Ein *Struct* mehrere *Traits*

```
1    impl Visit for Regular {
2        ...
3    }
4    impl Rule for Regular {
5        ...
6    }
```

Hier implementiert das *Struct* `Regular` aus Codeblock 1.8 die *Traits* `Visit` und `Rule`.

Enums und Pattern Matching

Enums und *Pattern Matching* sind in Rust eng miteinander verknüpft. *Pattern Matching* ist eine Möglichkeit, um unter anderem den Typ eines *Enums* zu prüfen und entsprechend zu reagieren. *Pattern Matching* ist in Rust ein flexibles Mittel, auf unterschiedlichste Patterns zu reagieren. Außerdem hat es die Besonderheit, dass immer alle Fälle abgedeckt sein müssen. Durch die erzwungene Abdeckung aller Fälle ist es nicht möglich, dass eine *Enum*-Variante nicht abgedeckt wird. *Enums* haben in Rust die Eigenschaft, dass sie Daten enthalten können. In Rust ist es also möglich, dass ein *Enum* verschiedene *Structs* enthält, die wieder Daten enthalten. Eines der bekanntesten *Enums* in Rust ist das `Option`-*Enum* (siehe Codeblock 1.10).

Codeblock 1.10: Das `Option` *Enum*

```
1    enum Option<T> {
2        None,
3        Some(T),
4    }
```

Es hat zwei Varianten, `None` und `Some(T)`. Dadruch ist es möglich, auf die enthaltenen Daten in `Some(T)` zuzugreifen oder auf die Abwesenheit von Daten zu reagieren. Das `Option`-*Enum* ist Rust's Möglichkeit, um mit *null*-Werten umzugehen. In Rust stellt der *Compiler* sicher, dass das `Option`-*Enum* immer abgefragt wird, bevor auf die enthaltenen Daten zugegriffen werden kann. So schafft es Rust mit der Abwesenheit von Werten umzugehen, ohne dafür

null-Werte zu verwenden. Diese zu *null pointer exceptions* führen können und vermieden werden sollten [42].
Ein weiterer Vorteil von *Enums* ist, dass sie beispielsweise Baumstrukturen abbilden können. Dies ist dadurch möglich, weil sie Daten enthalten können. So lassen sich *Enums* und *Structs* kombinieren, um komplexe Datenstrukturen zu erstellen.

2 Spezifikation

Im folgenden Kapitel sollen die Anforderungen an das Werkzeug, das im Rahmen dieser Arbeit entwickelt wird, definiert werden. Diese Anforderungen dienen als Grundlage für die Implementierung. Es soll ein Werkzeug zur statischen Programmanalyse entwickelt werden, das Quellcode auf Energieeffizienz prüft. Das Werkzeug soll in der Lage sein, verschiedene Arten von Energieeffizienzproblemen zu erkennen und dem Nutzer entsprechende Verbesserungsvorschläge geben.

2.1 Funktionale Anforderungen

Es soll eine Website entwickelt werden, auf der Nutzer ihren Quellcode einreichen können. Die Website soll in allen gängigen Browsern lauffähig sein. Das Werkzeug soll in der Lage sein, mit statischer Programmanalyse Energieeffizienzprobleme im Quellcode zu erkennen.

Eingabe:

- Nutzer können ihren Quellcode über ein Textfeld einreichen
- es kann mit einem Dropdown-Menü zwischen den Programmiersprachen *JavaScript*, *HTML* und *CSS* gewählt werden
  - für jede Programmiersprache gibt es unterschiedliche Regeln (siehe Abschnitt 2.1.1), die auf den Quellcode angewendet werden können
  - diese Regeln können mit einer Checkbox aktiviert oder deaktiviert werden
- mit einem Button kann der Quellcode analysiert werden

Diese Anforderungen sind in Abbildung 2.1 exemplarisch dargestellt.

Abbildung 2.1: Website mit Quellcode im Textfeld, Programmiersprache und Regel Auswahl

Ausgabe:

- der Quellcode wird in einem Textfeld angezeigt
  - die Zeilen, die ein Energieeffizienzproblem enthalten, werden markiert
  - die markierten Zeilen werden farblich hervorgehoben
  - Zeilen sind nummeriert
- eine tabellarische Übersicht zeigt die gefundenen Probleme
  - in der Tabelle werden die Schwere des Problems, die Zeile, die Spalte, der Typ des Problems und ein Verbesserungshinweis angezeigt

In Abbildung 2.2 sind die Ergebnisse einer Eingabe exemplarisch dargestellt.

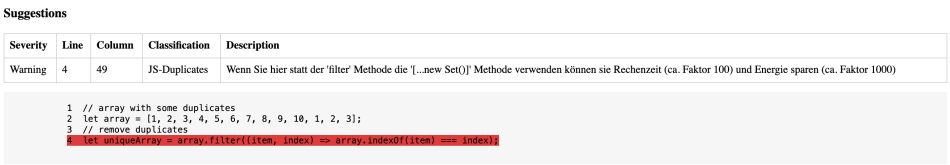

Abbildung 2.2: Darstellung der Ergebnisse der statischen Programmanalyse

2.1.1 Regeln

Der Quellcode kann mittels statischer Programmanalyse auf verschiedene Energieeffizienzprobleme geprüft werden. Die Regeln, die auf den Quellcode angewendet werden, sind hier beschrieben. Es sind Regeln, die sich auf das Kapitel 1.1 beziehen. Die dort vorgestellten Prinzipien dienen hier als Grundlage für die Regeln. Die Regeln sind in den *Web Sustainability Guidelines* (*WSG*) spezifiziert [3]. Sie sollen durch statische Programmanalyse auf ihre Einhaltung überprüft werden.

2.1.2 Lazy Loading

In der Kategorie 2.15 der *WSG* „Take a More Sustainable Approach to Image Assets“ (Sektion 2.15) geht es um den allgemeinen Umgang mit Bildern. Bilddaten verursachen einen großen Teil des Datenvolumens einer Website. Deshalb ist es wichtig, dass Bilder nur dann geladen werden, wenn sie auch benötigt werden. Wenn sie geladen werden, sollten sie möglichst klein sein.
Die Methode des *Lazy Loadings* ist eine Möglichkeit, Bilder nur dann zu laden, wenn sie auch benötigt werden. Das Erfolgskriterium wird in UX15-3: Include Lazy Loading With Images That Load Below-the-Fold spezifiziert. Hier wird nahegelegt, dass Bilder, die unterhalb des sichtbaren Bereichs der Website liegen, erst dann geladen werden, wenn sie in den sichtbaren Bereich gelangen. Es wird jedoch besprochen, dass das *Lazy Loading Flag* scheinbar keine Nachteile hat und immer gesetzt werden kann. Dies vereinfacht die Implementierung der *Lazy Loading*-Regel, da nicht zwischen Bildern im sichtbaren und unsichtbaren Bereich unterschieden werden muss. Auch das Testen der Regel wird dadurch vereinfacht, dass nur ein Kriterium getestet werden muss. Es ist lediglich entscheidend, ob das *Lazy Loading Flag* gesetzt ist. Aufwand und Auswirkung der Regel wurden als gering eingestuft [3].

2.1.3 Minify Code

In der Kategorie 3.2 der *WSG* „Minify Your HTML, CSS, and JavaScript“ (Sektion 3.2) geht es um das Minimieren von Code. *Minifying* ist ein Prozess, bei dem der Code verkleinert wird, indem unnötige Zeichen entfernt oder Funktions- und Variablennamen verkürzt werden. Mittels *Minifying* wird die Dateigröße verringert und dadurch auch das Datenvolumen beim Senden an den Nutzer reduziert. Die Erfolgskriterien nach WD02-1: Minify Your Front-End Code if It Is Public Facing lauten:

1. Prüfen, ob der Code minimiert werden kann.
2. Prüfen, ob es sich um öffentlich zugänglichen Code handelt.
3. Prüfen, ob Code-Obfuskation[1] verwendet wurde. Wenn nicht, dann minimieren.

Alle drei Kriterien müssen erfüllt sein, damit die Regel als erfolgreich angesehen wird. Auch hier wurden Aufwand und Auswirkung der Regel als gering eingestuft [3].

[1] bezeichnet den Vorgang, Quellcode bei gleicher Funktionsweise für Menschen möglichst unverständlich zu machen [43]

2.1.4 Vorteilhaftes JavaScript

In der Kategorie 3.15 der *WSG* „Use Beneficial JavaScript and Its APIs“ (Sektion 3.15) geht es um die Effizienz von Quellcode in JavaScript und Benutzer-Schnittstellen. Hier soll die Nachhaltigkeit durch Zugänglichkeit und Performance verbessert werden. Die Erfolgskriterien nach WD15-1: Optimize a Codebase Through Rewriting for Performance lauten:

1. Prüfen, ob der bestehende Code optimiert werden kann.
2. Prüfen, ob die verwendeten Werkzeuge nachhaltig und / oder performant sind.
3. Prüfen, ob JavaScript-Frameworks durch performantere Alternativen ersetzt werden können.
4. Prüfen, ob Frameworks ersetzt wurden.

Auch hier müssen alle Kriterien erfüllt sein, damit die Regel als erfolgreich umgesetzt angesehen wird. Der Aufwand wird als *mittel* und die Auswirkung als *hoch* eingestuft.
Um diese Regel weiter zu konkretisieren, soll die Methode *Filter* zum Entfernen von Duplikaten aus Arrays in JavaScript durch die Methode *Set* ersetzt werden. Hierzu wurde ein Benchmark erstellt, welches beide Methoden vergleicht:
In der ersten Abbildung 2.3 ist die Methode *Filter* aus Codeblock 4.2 dargestellt.

Codeblock 2.1: *Filter*-Methode zum Entfernen von Duplikaten aus einem Array

```
1    let array = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10];
2    let uniqueArray = array.filter((item, index) => array.indexOf(item) === index);
```

Mit der `filter`-Methode wird über jedes Element des Arrays iteriert und geprüft, ob der Index des Elements gleich dem Index des ersten Vorkommens des Elements ist. Ist dies der Fall, wird das Element in das neue Array kopiert. Kommt ein Element mehrmals vor, wird es nicht kopiert, da ab dem zweiten Vorkommen der Index nicht mit dem Wert der `indexOf`-Methode übereinstimmt. So bleiben am Ende nur die Elemente übrig, die zuerst im Array vorkommen und es entsteht ein Array ohne Duplikate. Für dieses Beispiel wird ein Array mit 100.000 Einträgen verwendet. Das Ergebnis ist in Abbildung 2.3 dargestellt. Die Laufzeit war hier durchschnittlich 2,2 Sekunden und der Energieverbrauch für den Methodenaufruf betrug 2,6mWh (Milliwattstunden).

In der zweiten Abbildung 2.4 ist die Methode *Set* in Codeblock 2.2 dargestellt.

Codeblock 2.2: *Set*-Methode zum Entfernen von Duplikaten aus einem Array

```
1    let array = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10];
2    let uniqueArray = [...new Set(array)];
```

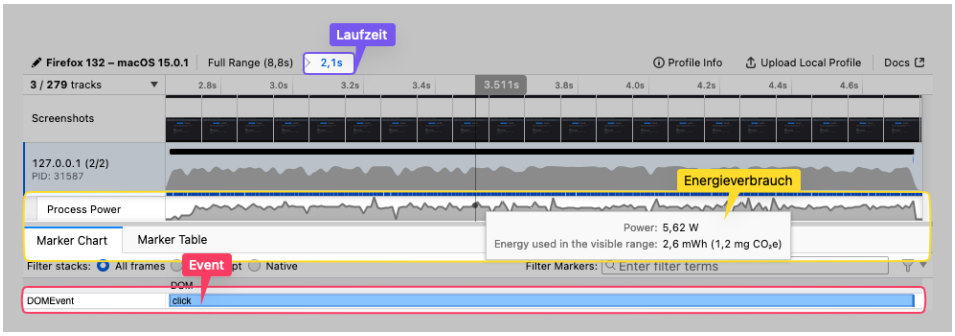

Abbildung 2.3: *Filter*-Methode zum Entfernen von Duplikaten aus einem Array mit 100.000 Einträgen

Mit der `new Set`-Methode wird ein neues *Set* aus dem Array erstellt, welches dann wieder in ein Array verwandelt wird. Da ein *Set* nur einzigartige Elemente speichert, werden alle Duplikate entfernt. Das Ergebnis ist in Abbildung 2.4 dargestellt. Die Laufzeit lag hier bei durchschnittlich 17ms und der Energieverbrauch für den Methodenaufruf betrug ca. 1µWh (Mikrowattstunden).

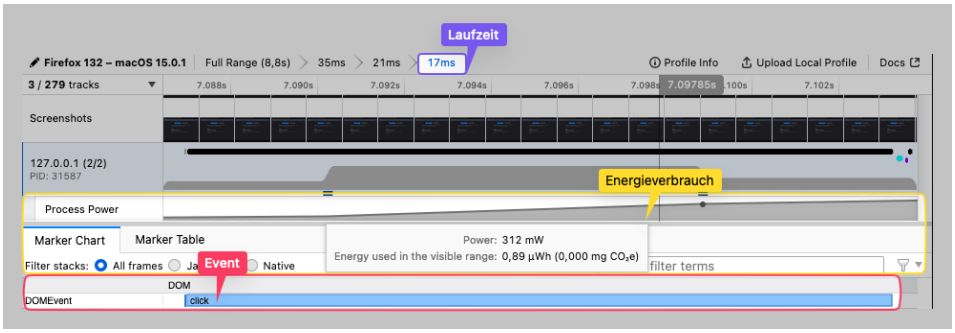

Abbildung 2.4: *Set*-Methode zum Entfernen von Duplikaten aus einem Array mit 100.000 Einträgen

Diese Unterschiede in Laufzeit und Energieverbrauch sind signifikant. In Bezug auf die Laufzeit ist die *Set*-Methode ca. 130-mal schneller als die *Filter*-Methode. Beim Energieverbrauch ist die *Set*-Methode rund 2.600-mal effizienter. Diese Werte sind von Messumgebungen abhängig und können stark variieren. Aber die Tendenz, dass die *Set*-Methode effizienter ist, bleibt bestehen. Deshalb sollte die Methode *Set* anstelle der Methode *Filter* verwendet werden, um Duplikate aus Arrays zu entfernen. Dies soll durch das Werkzeug überprüft werden.

2.2 Nicht-funktionale Anforderungen

Die nicht-funktionalen Anforderungen an das Werkzeug orientieren sich an den Prinzipien des nachhaltigen Webdesigns. Diese Prinzipien sind in Kapitel 1.1 beschrieben. Daraus ergeben sich folgende nicht-funktionale Anforderungen:

1. Sauberkeit
2. Effizienz
3. Offenheit
4. Ehrlichkeit
5. Regeneration
6. Resilienz

Des Weiteren soll das Werkzeug einfach zu bedienen sein. Auch die Erweiterbarkeit des Werkzeugs ist ein wichtiger Punkt, da es große Anzahl an Regeln gibt, die noch implementiert werden könnten. Dies gilt für JavaScript als auch für HTML, CSS und andere Programmiersprachen.

3 Entwurf

In diesem Kapitel wird der Entwurf der Anwendung beschrieben. Es wird auf die Architektur, die Technologien und die Struktur der Anwendung eingegangen. Außerdem wird erläutert, warum bestimmte Technologien und Architekturen gewählt wurden und wie die Anwendung aufgebaut ist.

3.1 Client-Server-Modell

Es wurde sich für ein Client-Server-Modell entschieden, da die Anwendung auf einem Server läuft und Methoden von einem Client aufgerufen werden. Der Client ist hier das Endgerät, auf dem die Website aufgerufen wird. Über die Website können Requests an den Server gesendet werden, der diese verarbeitet und eine Antwort zurückgibt (siehe Abbildung 3.1). So ist das Werkzeug unabhägig von einer IDE und kann von jedem Gerät mit einem Browser aufgerufen werden. Es wäre denkbar, das Werkzeug auch als Plugin für eine IDE zu entwickeln. Hier müsste das Plugin die Anfragen an das Backend senden und die Antwort verarbeiten.
Es muss keine Software installiert werden und es wird nur ein Browser benötigt. Die Anwendung ist also plattformunabhängig und kann auf jedem Gerät mit einem Browser aufgerufen werden. So werden die nicht-funktionalen Anforderungen der Offenheit und der leichten Bedienbarkeit unterstützt.

3.2 Remote Procedure Call statt REST

Es wurde sich für eine *Remote Procedure Call (RPC)*-Schnittstelle entschieden, da es sich hierbei um eine einfache Möglichkeit handelt, Methoden auf einem Server aufzurufen. Eine *RPC*-Schnittstelle bietet sich an, wenn keine Daten persistiert werden müssen und nur Operationen auf Eingaben ausgeführt werden. Es wird nur mit `GET`- und `POST`-Requests gearbeitet, da entweder Daten abgerufen werden (`GET`) oder Operationen mit Parametern aufgerufen werden (`POST`). Wenn Daten persistiert werden müssen, bietet sich eine REST Schnittstelle an, da hier noch `UPDATE`- und `DELETE`-Operationen hinzukommen, die es ermöglichen bestehende Daten zu verändern. In Abbildung 3.2 ist die Interaktion zwischen Client und Server dargestellt.

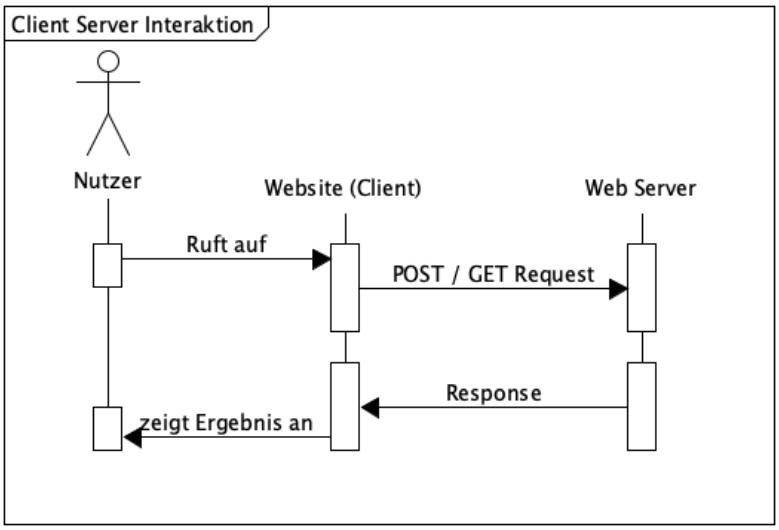

Abbildung 3.1: Client-Server Interaktion

Der *Client Stub* ist die Website, die HTTP-Requests mit einem JSON-Body an den Server (*Server Stub*) sendet. Der Server (*Server Stub*) macht aus dem JSON-Body ein Objekt und führt die Operation auf diesem aus. Das Ergebnis wird hier jedoch nicht als JSON zurückgegeben, sondern als HTML-Template, welches dann von der Website angezeigt wird. Es gibt also keine externe *Run-Time Library*, wie in der Abbildung 3.2 dargestellt. Stattdessen wurde eine eigene Schnittstelle implementiert. Die Interaktion ist in Abbildung 3.3 für den `evaluateCode`-

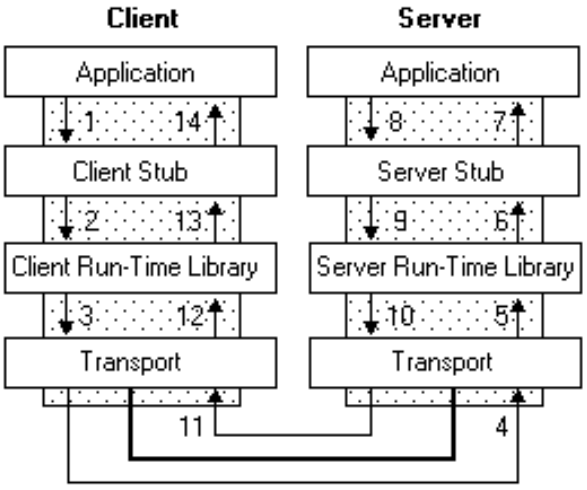

Abbildung 3.2: RPC Interaktion [44]

Endpunkt dargestellt. Aus dem HTML-Form wird ein JSON-Objekt erstellt und an den Server gesendet. Der Server ruft die `evaluateCode`-Methode mit den Parametern aus dem JSON Objekt

auf und gibt das Ergebnis zurück. Das Ergebnis ist ein Vektor aus `Result`-Objekten, die dann in ein HTML-Template eingefügt werden. Dieses wird dann an den Client zurückgegeben und dort gerendert.

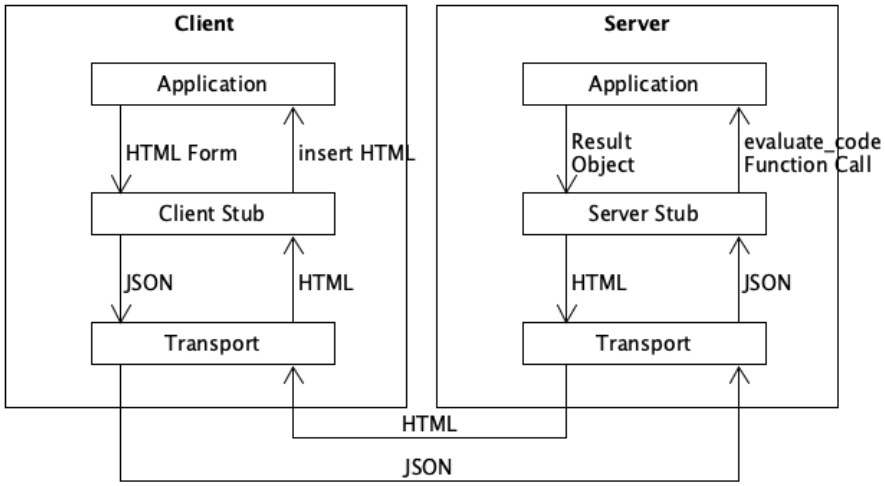

Abbildung 3.3: RPC-Interaktion für den `evaluateCode`-Endpunkt

3.3 Datenbank

Für die hier entwickelte Anwendung wird keine Datenbank benötigt, da keine Daten persistiert werden müssen. Auch Regeln werden nicht aus einer Datenbank geladen oder dort gespeichert, sondern sind im Code festgelegt. Nutzer können keine weiteren Regeln hinzufügen und so ist es nicht notwendig, diese in einer Datenbank zu speichern. Neue Regeln werden von Entwicklern im Code hinzugefügt. Es ist lediglich möglich, aus den vorhandenen Regeln auszuwählen und diese auf den Code anzuwenden. Außerdem gibt es keine Nutzer, die sich anmelden müssen.

3.4 Technologien

In diesem Abschnitt wird die Auswahl der Technologien für das Frontend und Backend erläutert.

3.4.1 Rust

Es wurde sich für die Programmiersprache Rust entschieden, da diese eine hohe Performance bietet und ein reiches Ökosystem an Libraries hat. Wie im Abschnitt „Energieverbrauch auf Serverseite“ 1.2.5 gezeigt, ist Rust nach C die Sprache mit dem geringsten Energieverbrauch und der höchsten Performance. Durch die Verwendung von Rust wird die Energieeffizienz und Performance des Servers sichergestellt. Dies hilft, die nicht-funktionale Anforderung der Effizienz zu erfüllen. Da Rust außerdem eine sichere und robuste Sprache ist, wird auch die nicht-funktionale Anforderung der Resilienz unterstützt.
Rust bietet eine gute Unterstützung für Webserver und Webanwendungen mit *Crates* wie *Axum* [45] oder *Actix* [46]. Auch für den Umgang mit JavaScript-Quellcode gibt es das *Crate OXC* [27], welches es ermöglicht, JavaScript-Quellcode zu analysieren.

3.4.2 Axum

Als Webserver wurde *Axum* gewählt, da dieser auf Rust basiert und eine gute Unterstützung für Webanwendungen bietet [45]. Es ist ein leichtgewichtiger Webserver, der es ermöglicht, schnell und einfach Webserver aufzusetzen. *Axum* wurde gewählt, da es ein Framework mit guter Dokumentation ist und von einem, in der Rust-Community anerkannten, Team entwickelt wird. Auch *Axum* ist auf Performance und Effizienz ausgelegt und so wird die nicht-funktionale Anforderung der Effizienz auch hier gestützt [47].

3.4.3 HTMX und Askama

HTMX wurde gewählt, da es eine einfache Möglichkeit bietet, dynamische Inhalte auf einer Website zu laden, ohne ein großes JavaScript-Framework zu verwenden.[1] Dadruch wird die nicht-funktionale Anforderung der Resilienz und der Effizienz unterstützt. Außerdem muss kein JavaScript geschrieben werden, da *HTMX* mit HTML-Attributen arbeitet. Es bietet sich besonders für kleine leichtgewichtige Websites an, die keine komplexen Interaktionen benötigen. Bei *HTMX* handelt es sich um einen *HTML Over The Wire*-Ansatz. Hierbei wird nur HTML übertragen, welches dann vom Browser gerendert wird. Im Falle von *HTMX* wird

[1] *HTMX* ist 67% kleiner als die React Bibliothek [48].

es in das bestehende HTML-Dokument eingefügt. Dadurch ist es möglich, dynamisch Inhalte bereitzustellen, ohne die Seite neu zu laden. Außerdem wird das Endgerät so nur minimal belastet, da nur sehr wenig JavaScript ausgeführt wird. Die hauptsächliche Arbeit passiert im Backend, welches hier auf Rust basiert und so in einer sicheren und performanten Umgebung ausgeführt wird.
Die *Askama Template Engine* wurde gewählt, da sie eine einfache Möglichkeit bietet, HTML-Templates zu erstellen. *Askama* bietet die Möglichkeit, dass *Structs* in den HTML-Template-Feldern erkannt werden und andersrum. Stimmt die Struktur des *Structs* nicht mit den Feldnamen im HTML-Template überein, wird ein Fehler zur Kompilierzeit geworfen. Dies hilft, die nicht-funktionale Anforderung der Resilienz zu erfüllen, da Fehler zur Kompilierzeit erkannt werden und so nicht zur Laufzeit auftreten können. Außerdem wird die nicht-funktionale Anforderung der Wartbarkeit unterstützt, da die Fehlererkennung die Wartung der Anwendung erleichtert.

3.5 Backend Architektur

Das Backend besteht aus mehreren Modulen. Die Hauptkomponenten sind der Webserver mit den Endpunkten und das Model mit der Logik. Es gibt Endpunkte, die die Regeln laden und an den Client senden und Endpunkte, die die Regeln auf den Code anwenden und das Ergebnis zurückgeben. Dies wird durch das Aufrufen von Methoden auf dem Model durch die Endpunkte erreicht. Die Endpunkte nutzen also die Schnittstelle des Models, wie in Abbildung 3.4 zu sehen ist. Durch diese Trennung ist die Anwendung leichter zu warten und

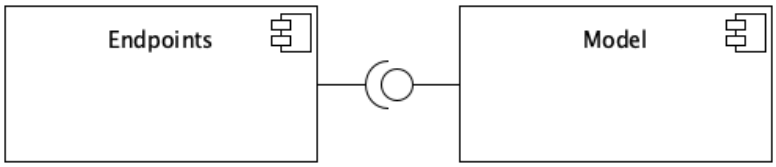

Abbildung 3.4: Endpoints und Model Komponente [44]

zu erweitern. Sie wird übersichtlicher und neue Funktionalitäten können leichter hinzugefügt werden, da sie einen festen Ort haben. Dies unterstützt die nicht-funktionale Anforderung der Wartbarkeit und der Erweiterbarkeit.

3.5.1 Endpunkte

Die Aufgabe der Endpunkte ist es, die Anfragen des Clients entgegenzunehmen und an das Model weiterzuleiten. Außerdem werden hier die HTML-Templates erstellt und an den Client zurückgegeben. Dazu müssen die Ergebnisse des Models ausgewertet werden und in ein HTML-Template eingefügt werden, wie bereits in Abbildung 3.3 dargestellt. Dieses wird dann an den Client zurückgegeben und dort gerendert.

3.5.2 Model

Das Model hat nach außen zwei Schnittstellen, die von den Endpunkten aufgerufen werden. Es können Regeln für die entsprechenden Programmiersprachen geladen werden, damit aus diesen ausgewählt werden kann.
Außerdem können die Regeln auf den Code angewendet werden. Dabei wird der Code, die Programmiersprache und die Regeln an das Model übergeben. Als Ergebnis kommt eine Liste mit den gefundenen Verbesserungen zurück.
Diese beiden Schnittstellen-Aufrufe sind als Sequenzdiagramm in den Abbildungen 3.5 und 3.6 dargestellt. In Abbildung 3.5 wird gezeigt, wie die Regeln geladen werden. Hier wird als

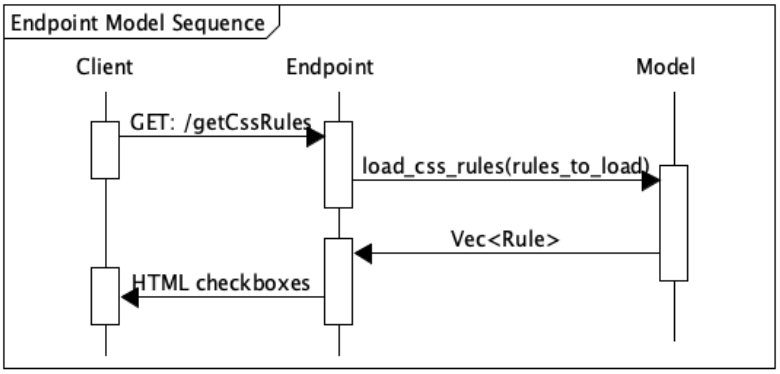

Abbildung 3.5: Regeln laden die dann als *Checkboxes* dargestellt werden (siehe auch 2.1)

Antwort eine Liste von Regeln zurückgeben.
In Abbildung 3.6 wird gezeigt, wie die Schnittstelle zum Anwenden der Regeln auf den Code funktioniert. Hier wird der Code, wie auch die Programmiersprache, als *String* und die Regeln die angewendet werden sollen, als Liste von *Strings* übergeben. Als Antwort kommt eine Liste von Ergebnissen für die jeweiligen Zeilen, in denen etwas gefunden wurde, zurück.

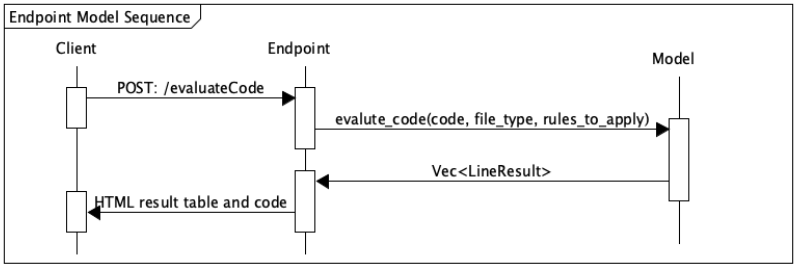

Abbildung 3.6: Quellcode einsenden und Ergebnis bekommen (siehe auch 2.2)

3.5.3 Regeln

Die Regeln sollen über ein *Trait* eingerichtet werden. Es sind bestimmte Methoden vorgesehen, die eine Regel implementieren muss (siehe Abbildung 3.7). In der `handler_method` wird über die ausgewählten Regeln iteriert und diese auf den Code mittels `apply`-Methode angewendet. Die Verwendung eines *Traits* ermöglicht es, neue Regeln einfach hinzuzufügen. Es stellt sicher,

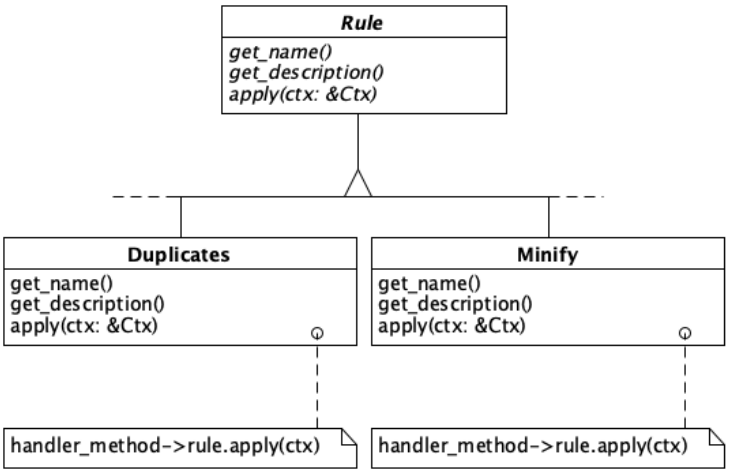

Abbildung 3.7: Das `Rule`-*Trait*

dass die Regeln die gleichen Operationen implementieren und so die gleiche Struktur haben. Dies garantiert dann, dass beim Iterieren über die Regeln immer die gleichen Operationen aufgerufen werden können. Auch das hilft, die nicht-funktionale Anforderung der Wartbarkeit zu erfüllen, da neue Regeln einfach hinzugefügt werden können und so die Anwendung leicht erweitert werden kann.

3.5.4 Programmanalyse

Für die Analyse des Codes wird mit statischer Programmanalyse gearbeitet. Das `Rule`-*Trait* ist für alle Programmiersprachen gleich. Es soll für alle Programmiersprachen implementiert werden.
Wie mit dem Quellcode verfahren wird, entscheidet sich in der Implementierung. Die Regeln werden für die unterschiedlichen Programmiersprachen in *Modulen* zusammengefasst (siehe Abbildung 3.8). In den `*.rs`-Dateien werden die Regeln implementiert und in den

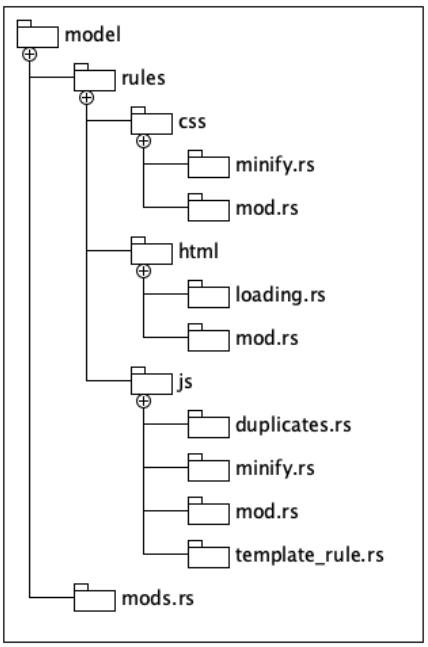

Abbildung 3.8: Modulstruktur

`mod.rs`-Dateien exportiert. So sollen Regeln einfach hinzufügbar sein, indem für die jeweilige Programmiersprache eine neue Datei erstellt wird. Hier wird dann das `Rule`-*Trait* implementiert und die neue Regel in der `mod.rs`-Datei exportiert.
Für die Programmanalyse in JavaScript wurde das *Crate OXC* verwendet. Es bietet die Möglichkeit, JavaScript-Quellcode zu parsen und einen *AST* zu erstellen. Der *AST* hat die *Root Node* `Program` (siehe Codeblock 1.2). Diese bietet den Einstiegspunkt für die Traversierung des *ASTs*. Traversiert wird der *AST* mithilfe des *Visitor Patterns*. Dazu muss das `Visit`-*Trait* implementiert werden. Hier können für jeden *Node Typ* die entsprechenden `visit_*`-Methoden überschrieben werden. So können die *Nodes* von den *Visitor*-Objekten besucht werden. *OXC* wurde gewählt, da es eine gute Unterstützung für JavaScript-Quellcode bietet und vollständig in Rust geschrieben ist. So müssen keine externen Bibliotheken verwendet werden und es kann direkt mit Rust gearbeitet werden. Die JavaScript Regeln müssen also nicht nur das

`Rule`-*Trait* implementieren, sondern auch das `Visit`-*Trait*, um mit dem *AST* arbeiten zu können (siehe Abbildung 3.9). Wie in der Spezifikation beschrieben, soll es eine Regel geben, die

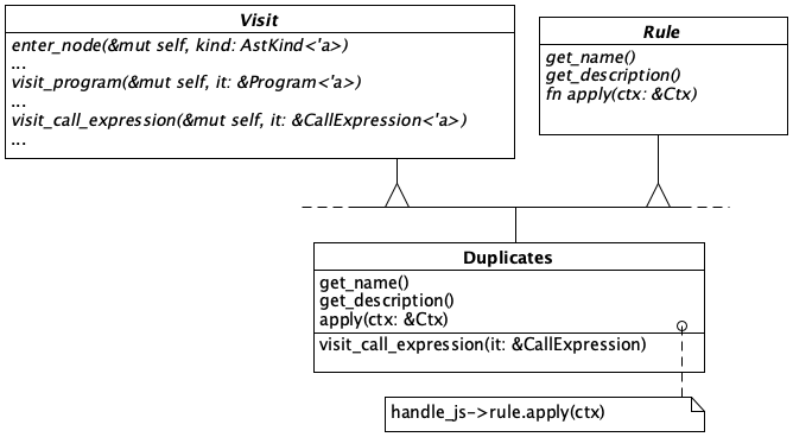

Abbildung 3.9: Konkrete Regel `Duplicates` implementiert das `Visit`-*Trait* und das `Rule`-*Trait*

das Entfernen von Duplikaten aus Arrays prüft. Die `Duplicates`-Regel aus Abbildung 3.9 soll diese Aufgabe erfüllen und überschreibt dafür die Methode `visit_call_expression`, da diese der Einstiegspunkt zum Methodenaufruf der *Filter*-Methode ist.

4 Umsetzung

In diesem Kapitel wird die Umsetzung des Werkzeugs beschrieben. Es wird auf die konkreten Entscheidungen eingegangen, die bei der Implementierung getroffen wurden.

4.1 Neue Regeln hinzufügen

Um eine neue Regel hinzuzufügen, muss sie an zwei Stellen im Projekt hinzugefügt werden. Zum einen muss ein neues Modul für die Regel erstellt werden, in dem die Logik der Regel implementiert wird. Dieses muss sich im Modul der Programmiersprache befinden, auf die die Regel angewendet werden soll (siehe Abbildung 3.8). Hier wird ein neues *Struct* erstellt, welches das *Trait* `Rule` implementiert. Es gibt eine `template_rule.rs`-Datei, die als Vorlage für neue Regeln dient und sich im JavaScript-Modul befindet. In dieser Datei sind alle notwendigen Methoden und Felder bereits vorgegeben, sodass nur die Logik der Regel implementiert werden muss.
Als zweiten Schritt muss die Regel noch der jeweiligen *Load*-Methode der entsprechenden Programmiersprache hinzugefügt werden. In diesem Fall handelt es sich um die Methode `load_js_rules`, die eine Liste von JavaScript-Regeln zurückgibt.
Regeln haben einen Namen und eine Beschreibung. Mit dem Namen werden die Regeln im Frontend dargestellt. Im Frontend ist die Regel als *Checkbox* sowie auch als HTML-Attribut (zu sehen in Codeblock 4.1) verankert. Dadurch ist es automatisch auch im *JSON*-Objekt, das an den Server gesendet wird, enthalten.
Das Askama-Template kann hier auf die Felder der *Structs* zugreifen, sodass nichts manuell eingetragen werden muss (siehe Codeblock 4.1). Es verwendet den Namen der Regel, um die *Checkbox* zu erstellen und den Wert der *Checkbox* zu setzen. So wird die Regel automatisch im Frontend dargestellt und kann ausgewählt werden.

Codeblock 4.1: Askama-Template für das Anzeigen und Auswählen der Regeln

```
1    {% for rule in checkboxes %}
2    <input type="checkbox" name="rule" value="{{rule.value}}" id="{{rule.value}}">
3    <label for="{{rule.value}}">{{rule.name}}</label>
4    {% endfor %}
```

In Codeblock 4.1 wird über die `checkboxes` iteriert, die alle anzuzeigenden Regeln enthalten. Dabei wird für jede Regel eine *Checkbox* erstellt und die Attribute werden automatisch gesetzt. Beim Hinzufügen einer neuen Regel nur dessen Name der Regel eingetragen werden. Der Rest wird automatisch generiert.
Des Weiteren wird der Name dann im Backend verwendet, um die ausgewählten Regeln zu filtern und auf den Quellcode anzuwenden. Dies passiert auch in der `load_js_rules`-Methode, die eine Liste von *Strings* mit den Namen der anzuwendenden Regeln bekommt und eine Liste von Regeln zurückgibt.
So ist es möglich, Regeln mit minimalen Anpassungen hinzuzufügen. Es muss lediglich eine Datei für die Regel erstellt und in der *Load*-Methode hinzugefügt werden. Damit das Projekt weiter wachsen kann und das Hinzufügen neuer Regeln dabei der wichtigste Schritt ist, wurde diese Struktur gewählt.

4.2 Regeln anwenden

Zum Anwenden der Regeln auf den Quellcode wird über diese iteriert und für jede die `apply`-Methode aufgerufen. Innerhalb der `apply`-Methode wird die Logik der Regel implementiert. Hier wäre es alternativ möglich gewesen, den Quellcode Zeile für Zeile bzw. in seiner alternativen Repräsentation zu traversieren und die Regeln auf die entsprechenden Stellen anzuwenden. So bräuchten die Regeln allerdings jeweils eine eigene Methode für jede Schnittstelle, die von der jeweiligen Quellcode-Repräsentation angeboten werden müsste. Eine `apply`-Methode hätte hier dann beispielsweise den Parameter *Node*, würde dann den *Node*-Typen prüfen und die Regel bei Übereinstimmung anwenden. Dadurch wäre es nicht mehr so einfach, neue Regeln hinzuzufügen, da unterschiedliche Programmiersprachen und auch gleiche Programmiersprachen eventuell unterschiedliche Schnittstellen und Code-Repräsentationen nutzen. Man könnte für jede Programmiersprache die jeweils gleiche Quellcode-Repräsentation wählen. Dann könnte man für jede Programmiersprache ein `Programming_Language_Rule`-*Trait* erstellen. So wäre es möglich, über die Quellcode-Repräsentation zu traversieren und die Regeln auf die entsprechenden Stellen anzuwenden. Dadurch würden die Regeln jedoch etwas an Flexibilität verlieren, da sie dann nicht mehr so spezifisch auf den jeweiligen Anwendungsfall zugeschnitten werden könnten. Das hier entwickelte Werkzeug soll eine Vielzahl unterschiedlicher Regeln für unterschiedliche Code-Repräsentationen und Programmiersprachen anbieten Deshalb wurde sich dafür entschieden, die Regeln auf den Quellcode bzw. die Quellcoderepräsentation anzuwenden.
Es wird für jede Regel die `apply`-Methode aufgerufen und der gesamte Quellcode bzw. die Quellcode-Repräsentation traversiert. Dies sollte in Bezug auf die Performance kein Problem darstellen Denn ob jede *Node* jede Regel befragt oder jede Regel jede *Node* überprüft, macht keinen Unterschied. Die Komplexität ist in beiden fällen $O(n^2)$.

Es ergibt sich jedoch ein Problem, wenn jede Regel selbst entscheidet, wie sie auf den Quellcode angewendet wird. Wenn lediglich der Quellcode als *String* übergeben wird, dann muss jede Regel ggf. den Quellcode in eine Repräsentation umwandeln, mit der sie arbeiten kann. Dadurch müsste die Umwandlung des Quellcodes in die entsprechende Repräsentation für jede Regel erneut durchgeführt werden. Dies wäre bei vielen Regeln, die auf der gleichen Repräsentation arbeiten, ineffizient.
Um dieses Problem zu umgehen, wurde ein *Context Enum* erstellt, welches den Quellcode in den entsprechenden Repräsentationen enthält. Wenn es sich bei dem *Input* um JavaScript handelt, wird der Quellcode einmalig in einen *AST* umgewandelt und zusammen mit dem Quellcode in einem *JavaScript Context* gespeichert. Dieses *Context*-Objekt kann dann von den Regeln verwendet werden, um auf den Quellcode und im Falle von JavaScript auch auf den *AST* zuzugreifen. Dadurch wird die Umwandlung des Quellcodes in die entsprechende Repräsentation nur einmal durchgeführt und die Regeln können auf den Quellcode zugreifen, ohne ihn jedes Mal umwandeln zu müssen.
Das *Enum* `Ctx` ist in Abbildung 4.1 zu sehen. Es besteht aus den *Enum*-Varianten *JavaScriptCtx*, *CssCtx* und *HtmlCtx*, welche wiederum die gleichnamigen konkreten Klassen enthalten. So kann das *Enum* in den Regeln verwendet werden, um auf den Quellcode zuzugreifen. Da immer nur eine Programmiersprache zur Anwendung kommt, bietet sich hier ein *Enum* an, da dies nur in einer Variante existieren kann. Für neue Programmiersprachen und neue Regeln muss ggf. das *Enum* und die jeweilige *Context*-Klasse erweitert werden. Die konkreten *Context*-Klassen sind auch auf Abbildung 4.1 zu sehen. Das *Context Enum* macht sich die Eigenschaft von Rust zunutze, dass *Enums* Daten, hier *Structs*, enthalten können. So können

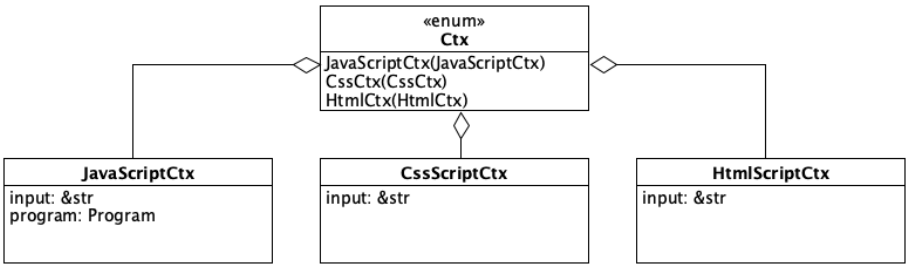

Abbildung 4.1: Das *Context Enum* welches an die Regeln übergeben wird

Regeln mit dem Quellcode je nach Bedarf verfahren. Sollte eine Quellcoderepräsentation häufiger verwendet werden, wäre es sinnvoll das entsprechende *Context Struct* zu erweitern. Dafür muss in dem *Context Struct* lediglich ein neues Feld hinzugefügt werden, welches die benötigte Coderepräsentation enthält. Außerdem muss bei der Initialisierung des *Context*-Objekts das neue Feld mit der Coderepräsentation befüllt werden. Im Anschluss kann das *Context*-Objekt an die Regeln übergeben werden, die dann auf die Felder zugreifen können.

4.3 Endpunkte

Die Endpunkte sind entsprechend der Spezifikation angelegt. Sie sind wie Methoden benannt, um eine RPC-Schnittstelle zu stellen (siehe Abbildungen Abbildung 3.5 und 3.6). Die Endpunkte `getCssRules`, `getJsRules`, `getHtmlRules` sind entsprechend der Programmiersprache benannt und mit einem `GET`-Request erreichbar. Sie geben eine Liste von verfügbaren Regeln zurück, die dann im Frontend als *Checkboxes* dargestellt werden. Hier hätte alternativ ein `GET`-Request mit einem *Query*-Parameter gewählt werden können, um Regeln zu laden. So hätte es nur einen Endpunkt zum Laden der Regeln gebraucht, der dann die Regeln für die entsprechende Programmiersprache zurückgibt. Es wurde sich jedoch für die Variante mit mehreren Endpunkten entschieden, damit die Regeln für die jeweilige Programmiersprache klarer getrennt sind. Außerdem haben *getter*-Methoden typischerweise keine Parameter, sondern geben nur Werte zurück. So sind die Endpunkte Methodenaufrufe im Backend, die die Regeln aus den entsprechenden Modulen laden (siehe Abbildung 3.8). Dies ist dann im Einklang mit der RPC-Schnittstelle, bei der es typisch ist, Methoden *remote* aufzurufen.
Es gibt nur einen `evaluateCode`-Endpunkt, der mit einem `POST`-Request und einem *JSON*-Objekt aufgerufen wird. Das *JSON*-Objekt enthält den Quellcode, die Programmiersprache und die ausgewählten Regeln. Hier wurde sich gegen eine Variante mit mehreren Endpunkten entschieden, da mehrere Parameter mitgegeben werden müssen, die dann in der Anwendung verarbeitet werden. Deshalb kann direkt der Parameter für die jeweilige Programmiersprache mitgegeben werden. Dies entspricht ebenfalls wieder der Backend-Struktur. Denn in der `evaluate_code`-Methode im Backend wird, je nach Programmiersprache, die entsprechende `handle`-Methode aufgerufen. Die `handle`-Methode erstellt das *Context*-Objekt, lädt die Regeln und ruft dann die `apply`-Methode auf den Regeln auf (siehe Abbildung 3.9). So sind die Endpunkte im Einklang mit der entworfenen Backend-Struktur.

4.4 Minify

Sowohl für CSS als auch für JavaScript wurde auf dem Quellcode *Minification* mithilfe von bereits bestehenden *Crates* angewendet. Das Ergebnis wird dann mit dem Quellcode verglichen und bei Unterschieden ein Hinweis erstellt. Der Hinweis beinhaltet, dass es sinnvoll ist, *Minification* zu verwenden, um die Ladezeit zu verbessern und weniger Daten zu versenden. Außerdem wird auf Tools hingewiesen, die die *Minification* automatisieren können.
Dies ist nicht in vollständigem Einklang mit der Spezifikation aus Abschnitt 2.1.3. Dort wird geprüft, ob *Code Obfuscation* vorliegt und ob der Quellcode öffentlich ist. Im Rahmen dieses Werkzeugs und des Anwendungsfalls ist das weder möglich noch sinnvoll. Wird CSS oder

JavaScript auf der Website auf *Minification* geprüft, so kann davon ausgegangen werden, dass der Quellcode öffentlich ist und keine *Code Obfuscation* stattgefunden hat.

4.5 HTML Lazy Loading

Hier wurde mithilfe eines *Crates* namens *Scraper* gearbeitet. *Scraper* baut aus dem HTML-Code einen Baum, in dem man mit einem *Selector* nach Tags suchen kann.
Es wurde eine Regel erstellt, die nach *Img* oder *Iframe*-Tags sucht. Bei diesen Tags wird dann überprüft, ob das `loading`-Attribut gesetzt ist. Ist es nicht gesetzt, wird vorgeschlagen es zu setzen, da es per Default auf *eager* gesetzt ist. Das heißt, das Bild wird sofort geladen, auch wenn es nicht im sichtbaren Bereich ist. Die Entscheidung, ob das Bild sofort oder erst im sichtbaren Bereich geladen wird, sollte jedoch bewusst getroffen werden. Es wird davon ausgegangen, dass diese Entscheidung nicht bewusst getroffen wurde, wenn das `loading`-Attribut nicht gesetzt ist.
Die *Minify* und die *HTML Lazy Loading*-Regel sind kleine Regeln. Diese wurden zuerst implementiert, um die Implementierung von Regeln zu testen. Sie sollen aber als Einstieg in die Thematik dienen und zeigen, dass auch mit geringem Aufwand schon Verbesserungen erreicht werden können.

4.6 Vorteilhaftes JavaScript

Für die Umsetzung der Regel zu vorteilhaftem JavaScript wurde die Duplikate-Entfernung aus Arrays gewählt (siehe Abschnitt 2.1.4).

4.6.1 Duplikate aus Arrays entfernen

Hier soll es um die Verwendung der *Filter*-Methode zur Entfernung von Duplikaten aus Arrays gehen. Passiert dies, soll sie durch die *Set*-Methode ersetzt werden. Um die *Filter*-Methode zur Entfernung von Duplikaten zu finden, müssen unterschiedliche Patterns gefunden werden. Es gibt unterschiedliche Schreibweisen für die *Filter*-Methode, die alle Duplikate aus einem Array entfernen. Die *Filter*-Methode kann mit einer *Function* oder einer *Arrow Function* aufgerufen werden. Außerdem kann die *Filter*-Methode mit zwei oder drei Parametern aufgerufen werden (`item`, `pos`, optionaler Parameter `self`). Innerhalb des *Bodies* der *Filter*-Methode kann mit '==' (*Equality*) oder '===' (*StrictEquality*) verglichen werden. Auch die Reihenfolge der Parameter kann im *Body* kann variieren. So entsteht eine Vielzahl von Patterns, die Duplikate aus einem Array entfernen.

Es gibt darüber hinaus eventuell noch weitere Patterns innerhalb der *Filter*-Methode, die Duplikate entfernen. Um diese nachträglich gut hinzufügen zu können, wurde die Regel so implementiert, dass sie leicht erweitert werden kann. Die Regel arbeitet auf einem *AST* und traversiert diesen, um die Patterns zu finden. Dafür wurde eine *Visitor*-Schnittstelle implementiert, die von *OXC* bereitgestellt wird.

4.6.2 Visitor Pattern am Beispiel von OXC

Um mit *OXC* zu arbeiten, muss das `Visit`-*Trait* implementiert werden. Dieses bietet eine Reihe von `visit`-Methoden an, die überschrieben werden können. Die Standardimplementierungen der `visit`-Methoden rufen alle jeweils nur eine `walk`-Methode auf, die wiederum die *Children* der *Nodes* traversiert. Alle `visit_*`-Methoden[1] haben also eine korrespondierende `walk_*`-Methode. Einstiegspunkt für die Traversierung ist ein Objekt, welches in *OXC* und auch sonst häufig `Program` genannt wird [29]. `Program` ist das Objekt, welches man vom *Parser* zurückbekommt und die *Root-Node* des *AST* ist (siehe auch Codeblock 1.2). Von hier wird mittels der `visit_program`-Methode die Traversierung durch das *Visitor*-Objekt gestartet. Jedes *Node*-Objekt hat fest definierte mögliche *Children*. Die `walk`-Methoden prüfen, welche der möglichen *Children* vorhanden sind und rufen dann die entsprechende `visit_*`-Methode auf ihnen auf. Das Verhalten der `visit_*`-Methoden kann durch das Überschreiben der `visit_*`-Methoden im `Visitor`-Objekt geändert werden. Ruft man in den überschriebenen `visit_*`-Methoden die `walk_*`-Methoden nicht auf, so wird die Traversierung nicht fortgesetzt. So kann durch das Überschreiben der `visit_*`-Methoden auf den *Nodes* gearbeitet werden.
Es gibt keine *Accept*-Methode, die auf den *Visitor*-Objekten aufgerufen wird. Stattdessen werden die konkreten `walk_*`-Methoden mit den *Node*-Objekten als Parameter aufgerufen. Die `walk_*`-Methoden sind also mit der *Accept*-Methode vergleichbar.

4.6.3 Implementierung der Visitor-Schnittstelle

Für das Auffinden der *Filter Patterns* wurde die `visit_call_expression`-Methode überschrieben. Es handelt sich bei der *CallExpression* um den *Node*-Typen, der für Methodenaufrufe steht. Der Einstiegspunkt für das *Filter Pattern* ist, unabhängig von den Unterschieden in der Schreibweise, immer die *Filter*-Methode. Deshalb ist die *CallExpression* als Einstiegspunkt hier optimal. Es werden alle *CallExpressions* besucht und nach den *Filter Patterns* durchsucht. Um die *Filter Patterns* zu finden, wird nach Unterbäumen im *AST* (dem ganzen Baum) gesucht, die entsprechend der *Filter Patterns* aufgebaut sind.

[1] der * ist hier immer Platzhalter für den Rest des Methodennames (z.B. hat die Methode `visit_call_expression` eine korrespondierende Methode namens walk_call_expression)

4.6.4 Filter Patterns

In Codeblock 4.2 ist eine Möglichkeit gezeigt, Duplikate aus einem Array zu entfernen. Auf dem Ursprungsarray wird die *Filter*-Methode aufgerufen und eine *Arrow Function* übergeben. Der `item`-Parameter ist das aktuelle Element und der `index`-Parameter ist der Index des aktuellen Elements. Die *Arrow Function* gibt *true* zurück, wenn das aktuelle Element das erste Element mit diesem Wert ist (`array.indexOf(item) === index`), ansonsten *false*. So gibt die *Filter*-Methode dann ein neues Array zurück, in dem alle Elemente enthalten sind, die das erste Element mit diesem Wert sind.

Codeblock 4.2: Ein exemplarisches Filter Pattern zum Filtern von Duplikaten aus Arrays

```
1    let uniqueArray = array.filter((item, index) => array.indexOf(item) === index);
```

In Abbildung 4.2 ist ein Ausschnitt eines Unterbaumes eines *AST* dargestellt, der das *Filter Pattern* aus Codeblock 4.2 darstellt. Es basiert auf dem *AST* aus *OXC* und zeigt die *CallExpression* und die enthaltene *ArrowFunction* mit ihren Bestandteilen.
In Rot markiert sind die *Leaves* des Baumes, die zusammen wieder auf den Quellcode zurück-

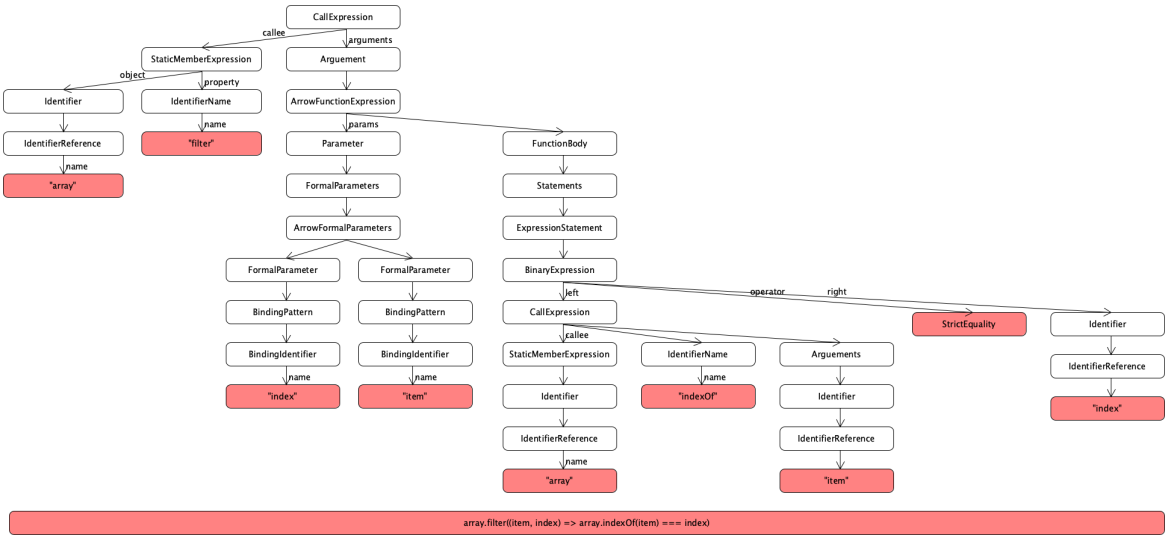

Abbildung 4.2: Ausschnitt eines Unterbaumes des *AST*, der das *Filter Pattern* darstellt und auf der Basis JSON-ASTs aus *OXC* erstellt wurde

geführt werden können. Mit der *CallExpression* als Einstiegspunkt kann dann der gesamte Unterbaum traversiert werden, um das *Filter Pattern* zu finden. Dazu wird *Node* für *Node* überprüft, ob es sich um den Baum aus Abbildung 4.2 handelt. Dabei ist das *Pattern Matching* in Rust zugleich hilfreich und mühsam. Es muss für jede *Node* explizit geprüft werden, ob sie existiert und den erwarteten Wert hat. Dadurch werden jedoch Fehler zur Laufzeit vermieden,

da der Compiler prüft, ob alle Fälle abgedeckt sind. *Nodes* müssen auf alle relevanten Eigenschaften überprüft werden, um sicherzustellen, dass es sich um das *Filter Pattern* handelt. Eine relevante Eigenschaft sind die *Identifier* der Parameter. Diese werden in der *Arrow Function* übergeben und danach im `indexOf(item) === index` Statement verwendet. Sie müssen übereinstimmen, damit das Pattern gefunden wird. Dies ist auch im Unterbaum-Ausschnitt in Abbildung 4.2 zu sehen. Dafür speichert die Regel die *Identifier* der Parameter und prüft sie später gegen die *Identifier*, die im `indexOf(item) === index` Statement verwendet werden. Entspricht eine *Node* nicht dem Unterbaum aus Abbildung 4.2, wird die Suche für den Unterbaum abgebrochen und die Traversierung des *AST* fortgesetzt.
Für diese Arbeit wurden weitere Variationen des *Filter Patterns* gefunden und implementiert. Es handelt sich um bisher insgesamt 24 Variationen, die sich durch die Kombination aus den Unterschieden in der Schreibweise der *Filter*-Methode ergeben. So wird der gesamte Unterbaum immer größer und komplexer, je mehr Variationen gefunden werden, die zum gleichen Ergebnis führen. Es geht also darum, semantisch äquivalente Patterns zu finden.
Gibt es für die Regel einen Weg durch den Unterbaum, wurde ein *Filter Pattern* gefunden.
Es wäre möglich, für jedes Pattern eine neue Regel zu erstellen. So wäre es leichter, neue Patterns hinzufügen. Denn dadruch müsste nur eine neue Regel erstellt werden, die das neue Pattern findet. Aktuell muss die Regel erweitert werden, um das neue Pattern zu finden. Dafür ist es notwendig, die entsprechende Stelle zu finden, an der das neue Pattern von den bisherigen abweicht. An dieser Stelle muss die Regeln dann erweitert werden. Um diesen Prozess zu vereinfachen, wurde die Regel so implementiert, dass die einzelnen, für die Patterns relevanten, *Nodes* in separaten Methoden behandelt werden. Außerdem gibt es für jedes Pattern Tests, die sicherstellen, dass das Pattern gefunden wird. Soll ein neues Pattern hinzugefügt werden, kann zuerst getestet werden, ob es bereits gefunden wird, indem ein Test für das Pattern geschrieben wird. Wird es nicht gefunden, muss die Regel erweitert werden, um das Pattern zu finden.

5 Fazit

Ziel der Arbeit war die Entwicklung eines Werkzeugs, welches mittels statischer Programmanalyse die Energieeffizienz von Websites verbessert. Dazu mussten zunächst die Grundlagen des nachhaltigen Webdesigns und des Energieverbrauchs von Websites erarbeitet werden. Es wurden einige der Vorschläge aus den *Web Sustainability Guidelines* umgesetzt. Die Umsetzung erfolgte, indem aus den Vorschlägen Regeln gemacht wurden. Diese Regeln wurden dann implementiert und mittels statischer Programmanalyse auf den Quellcode angewendet. So ist es nun möglich, die Vorschläge automatisiert zu überprüfen und Entwickler auf mögliche Verbesserungen hinzuweisen.
Auch das Werkzeug selbst sollte möglichst energieeffizient sein. Dafür wurde Rust als Programmiersprache gewählt, da sie eine hohe Performance bietet. Im Frontend wurde mit HTMX auch auf eine leichtgewichtige Bibliothek gesetzt, um Energieeffizienz zu gewährleisten.
Es ist also möglich, mit dem Werkzeug die Energieeffizienz von Websites zu verbessern und gleichzeitig energieeffizient zu arbeiten. Die automatisierte Überprüfung des Codes kann Entwicklern dabei helfen, energieeffizienten Code zu schreiben. Trotz der, bisher sehr wenigen, Anwendungsfälle gibt dies Anlass zur Hoffnung, dass Werkzeuge, wie das hier entwickelte, in Zukunft eine wichtige Rolle spielen könnten. Sollte an entsprechenden Werkzeugen gearbeitet werden, könnte dies dazu beitragen, den Energieverbrauch von Websites zu reduzieren. Motivation dafür gibt es genug. Denn Energieineffizienz geht fast immer mit Performance-Problemen einher. Selbst, wenn einem die Umwelt nicht am Herzen liegt, kann es sinnvoll sein, energieeffizienten Code zu schreiben. Dies kann Anreize schaffen, sich mit dem Thema auseinanderzusetzen und entsprechende Werkzeuge zu entwickeln.
Schwierigkeiten bei der Entwicklung des Werkzeuges waren die Einarbeitung in Rust und die Implementierung der statischen Programmanalyse. Rust zeigt häufig unerwartetes Verhalten, wenn man nicht mit der Sprache vertraut ist. Es gibt oft Fehlermeldungen zur Kompilierzeit, jedoch nahezu gar keine zur Laufzeit. Diese Fehlermeldungen sind hilfreich, aber oft auch schwierig zu verstehen, da sie Konzepte der Sprache voraussetzen.
Auch die Implementierung der statischen Programmanalyse war nicht einfach. *OXC* bietet zwar eine gute Dokumentation, aber wenige Beispiele. Es gibt keine *Crates* die *OXC* verwenden, außer *OXC* selbst. Daher war es schwierig, sich in die Funktionsweise von *OXC* einzuarbeiten.
Es gibt noch viele Vorschläge aus den *Web Sustainability Guidelines*, die implementiert werden

könnten. Auch die Implementierung der Vorschläge, die bereits als Regeln umgesetzt wurden, könnte verbessert werden.
Ein weiterer Schritt könnte auch das Anbieten des Werkzeugs als Plugin für IDEs sein. So könnten Entwickler direkt beim Schreiben des Codes auf mögliche Verbesserungen hingewiesen werden.
Auch eine tiefer gehende Analyse Gründe für Unterschiede in der Energieeffizienz könnte interessant sein. Hier wäre es spannend Implementierungs-Details zu vergleichen und zu analysieren, wo die Energieeffizienzunterschiede entstehen.

Literatur

[1] T. Greenwood, *Nachhaltiges-Webdesign*. Nürnberg: conopolist Verlag, 2021.

[2] T. W. W. W. Consortium. „The World Wide Web Consortium," besucht am 1. Aug. 2024. Adresse: https://www.w3.org/.

[3] T. F. Alexander Dawson. „Web Sustainability Guidelines (WSG) 1.0," besucht am 1. Aug. 2024. Adresse: https://w3c.github.io/sustyweb/#principles.

[4] S. W. D. C. Group. „A community group dedicated to creating sustainable websites.," besucht am 1. Aug. 2024. Adresse: https://www.w3.org/groups/cg/sustyweb/.

[5] T. W. W. W. Consortium. „About W3C web standards," besucht am 1. Aug. 2024. Adresse: https://www.w3.org/standards/about/#what-are-web-standards.

[6] T. G. Compact. „Who Cares Wins, Connecting Financial Markets to a Changing World," besucht am 24. Juli 2024. Adresse: https://www.unepfi.org/fileadmin/events/2004/stocks/who_cares_wins_global_compact_2004.pdf.

[7] J. Elkington, *Cannibals with Forks, The Triple Bottom Line of 21st Century Business*. Capstone, 1999.

[8] W. Digital. „Sustainable Web Manifesto," besucht am 1. Aug. 2024. Adresse: https://www.sustainablewebmanifesto.com.

[9] U. Franklin, *The Real World of Technology* (Massey lectures series). Anansi, 1999, ISBN: 9780887846366. Adresse: https://books.google.cz/books?id=LziQT3YS_2sC.

[10] Nature. „How to stop data centres from gobbling up the world's electricity," besucht am 29. Juli 2024. Adresse: https://www.nature.com/articles/d41586-018-06610-y.

[11] L. Belkhir und A. Elmeligi, „Assessing ICT global emissions footprint: Trends to 2040 I& recommendations," *Journal of Cleaner Production*, Jg. 177, S. 448–463, 3. Jan. 2018, ISSN: 0959-6526. DOI: https://doi.org/10.1016/j.jclepro.2017.12.239. Adresse: https://www.sciencedirect.com/science/article/pii/S095965261733233X.

[12] I. E. Agency. „Data Centres and Data Transmission Networks," besucht am 30. Juli 2024. Adresse: https://www.iea.org/energy-system/buildings/data-centres-and-data-transmission-networks#tracking.

[13] T. Independent. „Global warming, Data centres to consume three times as much energy in next decade, experts warn," besucht am 26. Juli 2024. Adresse: https://www.independent.co.uk/climate-change/news/global-warming-data-centres-to-consume-three-times-as-much-energy-in-next-decade-experts-warn-a6830086.html.

[14] W. Digital. „Why do estimates for internet energy consumption vary so drastically?" Besucht am 30. Juli 2024. Adresse: https://www.wholegraindigital.com/blog/website-energy-consumption/.

[15] S. W. Design. „Estimating Digital Emissions," besucht am 30. Juli 2024. Adresse: https://sustainablewebdesign.org/estimating-digital-emissions/.

[16] R. Pereira u. a., „Ranking programming languages by energy efficiency," *Science of Computer Programming*, Jg. 205, S. 102 609, 2021, ISSN: 0167-6423. DOI: https://doi.org/10.1016/j.scico.2021.102609. Adresse: https://www.sciencedirect.com/science/article/pii/S0167642321000022.

[17] Intel. „Running Average Power Limit (RAPL)," besucht am 31. Juli 2024. Adresse: https://www.intel.com/content/www/us/en/developer/articles/technical/software-security-guidance/advisory-guidance/running-average-power-limit-energy-reporting.html?wapkw=rapl.

[18] Mozilla. „Firefox Profiler," besucht am 31. Juli 2024. Adresse: https://profiler.firefox.com/docs/#/.

[19] Apple. „Timelines Tab," besucht am 31. Juli 2024. Adresse: https://webkit.org/web-inspector/timelines-tab/#cpu-timeline.

[20] A. Møller und M. I. Schwartzbach, *Static Program Analysis*. Department of Computer Science, Aarhus University, 2024.

[21] A. Gosain und G. Sharma, „A Survey of Dynamic Program Analysis Techniques and Tools," in *Proceedings of the 3rd International Conference on Frontiers of Intelligent Computing: Theory and Applications (FICTA) 2014*, S. C. Satapathy, B. N. Biswal, S. K. Udgata und J. Mandal, Hrsg., Springer International Publishing, 2015, S. 113–122.

[22] Microsoft. „Unit Testing: Testing the Inside," besucht am 22. Aug. 2024. Adresse: https://learn.microsoft.com/en-us/previous-versions/msp-n-p/jj159340(v=pandp.10).

[23] R. Nystrom. „Crafting Interpreters," besucht am 23. Aug. 2024. Adresse: https://craftinginterpreters.com/contents.html.

[24] L. M. University. „Static Analysis," besucht am 23. Aug. 2024. Adresse: https://cs.lmu.edu/~ray/notes/semanticanalysis/.

[25] J. Jones. „Abstract Syntax Tree Implementation Idioms," besucht am 5. Aug. 2024. Adresse: https://hillside.net/plop/plop2003/Papers/Jones-ImplementingASTs.pdf.

[26] E. International. „ECMAScript Language Specification,“ besucht am 30. Aug. 2024. Adresse: https://262.ecma-international.org/15.0.

[27] Boshen. „The JavaScript Oxidation Compiler, A collection of JavaScript tools written in Rust,“ besucht am 24. Juli 2024. Adresse: https://oxc.rs.

[28] F. Kling. „AST Explorer,“ besucht am 5. Aug. 2024. Adresse: https://astexplorer.net.

[29] Boshen. „Oxc-ast,“ besucht am 5. Aug. 2024. Adresse: https://oxc.rs/docs/learn/parser_in_rust/ast.html.

[30] e. a. Erich Gamma, *Design Patterns, Entwurfsmuster als Elemente wiederverwendbarer objektorientierter Software*. mitp, 2015.

[31] S. Overflow. „Stack Overflow Developer Survey 2023,“ besucht am 5. Aug. 2024. Adresse: https://survey.stackoverflow.co/2023/#section-admired-and-desired-programming-scripting-and-markup-languages.

[32] T. R. P. Language. „The Rust Programming Language,“ besucht am 5. Aug. 2024. Adresse: https://www.rust-lang.org/.

[33] T. R. P. Language. „Cargo,“ besucht am 5. Sep. 2024. Adresse: https://doc.rust-lang.org/cargo/.

[34] T. R. P. Language. „rustfmt,“ besucht am 5. Aug. 2024. Adresse: https://github.com/rust-lang/rustfmt.

[35] T. R. P. Language. „Clippy,“ besucht am 5. Aug. 2024. Adresse: https://doc.rust-lang.org/clippy/.

[36] T. R. P. Language. „Rust Ownership,“ besucht am 5. Aug. 2024. Adresse: https://doc.rust-lang.org/book/ch04-00-understanding-ownership.html.

[37] D. Dhurjati, S. Kowshik, V. Adve und C. Lattner, „Memory Safety without runtime checks or garbage collection,“ *SIGPLAN Not.*, Jg. 38, Nr. 7, S. 69–80, Juni 2003, ISSN: 0362-1340. DOI: 10.1145/780731.780743. Adresse: https://doi.org/10.1145/780731.780743.

[38] Microsoft. „Fundamentals of garbage collection,“ besucht am 5. Aug. 2024. Adresse: https://learn.microsoft.com/en-us/dotnet/standard/garbage-collection/fundamentals.

[39] OWASP. „Doubly freeing memory,“ besucht am 5. Aug. 2024. Adresse: https://owasp.org/www-community/vulnerabilities/Doubly_freeing_memory.

[40] C. Cimpanu. „Microsoft: 70 percent of all security bugs are Memory Safety issues,“ besucht am 5. Aug. 2024. Adresse: https://www.zdnet.com/article/microsoft-70-percent-of-all-security-bugs-are-memory-safety-issues/.

[41] T. R. P. Language. „crates.io,“ besucht am 5. Sep. 2024. Adresse: https://crates.io.

[42] T. Hoare. „Null References The Billion Dollar Mistake,“ besucht am 5. Sep. 2024. Adresse: https://www.infoq.com/presentations/Null-References-The-Billion-Dollar-Mistake-Tony-Hoare/.

[43] B. Lutkevich. „obfuscation,“ besucht am 5. Aug. 2024. Adresse: https://www.techtarget.com/searchsecurity/definition/obfuscation.

[44] Microsoft. „Funktionsweise von RPC,“ besucht am 5. Sep. 2024. Adresse: https://learn.microsoft.com/de-de/windows/win32/rpc/how-rpc-works.

[45] Axum. „Axum,“ besucht am 5. Sep. 2024. Adresse: https://github.com/tokio-rs/axum.

[46] Actix. „Actix,“ besucht am 8. Okt. 2024. Adresse: https://actix.rs.

[47] T. Empower. „Web Framework Benchmarks,“ besucht am 5. Sep. 2024. Adresse: https://www.techempower.com/benchmarks/#hw=ph&test=fortune&section=data-r22.

[48] htmx. „htmx,“ besucht am 5. Sep. 2024. Adresse: https://htmx.org.

Eigenständigkeitserklärung

Hiermit versichere ich, dass ich die vorliegende Bachelorarbeit mit dem Titel

Entwicklung eines Werkzeugs zur Reduzierung des Energieverbrauchs von Websites durch statische Code-Analyse

selbstständig und nur mit den angegebenen Hilfsmitteln verfasst habe. Alle Passagen, die ich wörtlich aus der Literatur oder aus anderen Quellen wie z. B. Internetseiten übernommen habe, habe ich deutlich als Zitat mit Angabe der Quelle kenntlich gemacht.

Hamburg, den