Universität Hamburg
DER FORSCHUNG | DER LEHRE | DER BILDUNG

Masterarbeit

Analyse der Verbreitung und automatisierten Erkennung relevanter Code Smells im Web Development

Janik Schröder

janik.schroeder@studium.uni-hamburg.de
MIN-Fakultät, Fachbereich Informatik
Studiengang Wirtschaftsinformatik
Matr.-Nr. 6930372

Erstgutachterin: Paula Rachow
Zweitgutachter: Univ.-Prof. Dr.-Ing. habil. Matthias Riebisch

Abgabe: 10.2022

Zusammenfassung

Aus der stets zunehmenden Komplexität von Web-Anwendungen resultiert die Notwendigkeit der Verbesserung von deren Code-Qualität. Ein anerkanntes Maß aus der „klassischen“ Software-Entwicklung für die Einschätzung der Code-Qualität ist das Vorhandensein von Code Smells. Bei der Übertragung dieser Thematik werden die Besonderheiten der Web-Entwicklung bisher zu sehr vernachlässigt. Insbesondere das Zusammenspiel der Komponenten HTML, CSS und JavaScript wird dabei missachtet – bisherige Arbeiten und Tools thematisieren fast ausschließlich JavaScript-Smells. Auch hinsichtlich der Verbreitung verschiedener Smells im Web fehlt es an aussagekräftigen Erkenntnissen. Diese könnten jedoch die Grundlage dafür schaffen, besonders stark verbreitete Smells zukünftig priorisiert zu behandeln und die Code-Qualität im Web dadurch nachhaltig zu verbessern.

In dieser Arbeit werden Erkenntnisse dieser Art gewonnen. Durch die Entwicklung des Smell-Identifikations-Tools *WebDeo* gelang es, die Verbreitung von 39 für die Web-Entwicklung relevanter Code Smells in den Sprachen HTML, CSS und JavaScript auf 10.255 Websites zu analysieren. Durch die inhaltliche Begleitung der Entwicklung werden Möglichkeiten und Grenzen des Ansatzes der statischen Code-Analyse zur Identifikation von Code Smells deutlich.

Die Ergebnisse zeigen, dass es extreme Unterschiede in der Verbreitung einzelner Smells gibt. Die Gültigkeit des *Paretoprinzips* hinsichtlich der Häufigkeit belegt den potenziellen Nutzen einer zukünftigen Priorisierung ausgewählter Smells. Außerdem zeigt diese Arbeit, dass der bestehende Fokus auf JavaScript-Smells in Anbetracht derer Verbreitung unbegründet ist. Eine ganzheitliche Betrachtung von Code Smells in der Web-Entwicklung – inklusive der Sprachen HTML und CSS – empfiehlt sich.

Durch die Entwicklung von *WebDeo* wird im Rahmen dieser Arbeit ein Tool präsentiert, welches diesen ganzheitlichen Ansatz nutzt. Neben HTML-, CSS- und JavaScript-Smells identifiziert *WebDeo* auch schlechte Praktiken, die sich aus der besonderen Kombination dieser Komponenten in der Web-Entwicklung ergeben. Durch die Bereitstellung eines Feedback-Reports kann *WebDeo* Entwickler*innen dabei unterstützen, Code Smells zu entfernen und sich den individuellen Stärken und Schwächen hinsichtlich der eigenen Code-Qualität bewusst zu werden.

Inhaltsverzeichnis

1. Einleitung

1.1. Motivation

Die Web-Entwicklung hat sich in den vergangenen Jahren fortlaufend und grundlegend gewandelt. Inzwischen haben sich interaktive Web-Entwicklungen („Web-Apps“), als Standard etabliert und die früher üblichen, statischen Informationswebsites verdrängt [Sar15]. Zu den charakteristischen Merkmalen von Web-Apps gehören etwa aufwändige funktionale Features, beeindruckende Designs und Effekte, die Möglichkeit der Speicherung und Verarbeitung von Daten sowie die fortlaufende Nutzer*innen-Interaktion, zum Beispiel durch die Möglichkeit Push-Benachrichtigungen zu versenden. Für die Unternehmen bieten Web-Apps inzwischen sogar deutliche Kosten- und Nutzen-Vorteile im Vergleich zu nativen Mobile Apps [Col19, BOM20]. Die beschriebene Entwicklung wird von Himanshu Sareen, Gründerin und Geschäftsführerin von Icreon Tech, als „Appifizierung der Standard-Website“ zusammengefasst.

> *„Mobile apps have skyrocketed expectations for what a high quality digital experience can be. And as a result,* ***the standard website has officially become app-ified*** *in an attempt for brands, organizations, and webmaster to maintain attention for the somewhat struggling desktop browser. “*
>
> — Himanshu Sareen *[Sar15]*

Mit der beschriebenen Entwicklung einhergehend steigen auch die Anforderungen an die Sicherheit, Bedienbarkeit und Stabilität von Websites [Moh20]. Zusammenfassend müssen moderne Web-Entwickler*innen in der Lage sein, die stark gestiegene Komplexität von Web-Anwendungen beherrschbar zu machen. Durch die fortlaufende Expansion des Webs [Intb, Inta] und aufgrund des Fakts, dass sogar Websites kleinster Unternehmen von der „Appifizierung“ inbegriffen sind [Sar15], nimmt die Relevanz dieser qualitativen Herausforderung weiter zu, da sie zusätzlich eine wachsende quantitative Dimension besitzt.

Je komplexer ein Software-Projekt ist, desto mehr wird eine hohe Code-Qualität zur Grundvoraussetzung zur erfolgreichen Durchführung von Änderungen. Durch eine gute Verständlichkeit des Codes wird sichergestellt, dass dieser schnell, risikoarm und von verschiedenen Entwickler*innen angepasst werden kann [Fow20].

Ein anerkannter Indikator zur Bewertung der Qualität und Verständlichkeit von Code sind „Code Smells“. Dabei handelt es sich um charakteristische Schwächen hinsichtlich der Code-Struktur, welche negative Auswirkungen auf die Verständlichkeit und somit auch auf die Anpassbarkeit des Codes haben können [JKA19, SS18]. Code Smells – häufig auch einfach als *Smells* abgekürzt – sind in der Fachliteratur mit vielen Beispielen

hinsichtlich ihrer Identifikation sowie Anleitungen zur angemessenen Überarbeitung des Codes beschrieben. Die Analyse vorliegender Code Smells eignet sich somit sehr gut, um die Qualität von Code einschätzen zu können und identifizierte Schwachstellen anschließend systematisch zu verbessern („Refactoring“) [SS18]. Code Smells kommt somit auch eine große Bedeutung hinsichtlich der Beherrschung der steigenden Komplexität in der Web-Entwicklung zu.

Die Web-Entwicklung ist stark durch das Zusammenspiel der browserseitig interpretierten Sprachen HTML (Hypertext Markup Language; Struktur), CSS (Cascading Style Sheets; Design) und JavaScript (Funktionalität) geprägt. Die gemeinsame Betrachtung der drei Sprachen und ihrer Interaktion untereinander ist daher unabdingbar. Dennoch fällt auf, dass es im Kontext der Themen Code Smells und Code-Qualität an diesem ganzheitlichen Blick mangelt, sowohl in der Forschung als auch in der Praxis. Ebenso wie in bestehenden wissenschaftlichen Arbeiten liegt der Fokus auch bei existierenden Software-Tools fast immer auf einer einzelnen Sprache – meistens JavaScript (⇒ Kapitel 3). Unklar bleibt, ob es berechtigte Gründe für diese Forschungslücke gibt.

Diese Arbeit verfolgt das Ziel zum Forschungsstand bezüglich Code Smells in der Web-Entwicklung und zu der Verbesserung der Software-Qualität im Web beizutragen. Durch eine groß angelegte Analyse der Verbreitung von Code Smells im Web sollen etwa Erkenntnisse dazu erlangt werden, ob Code Smells in JavaScript ein deutlich stärker verbreitetes Problem sind als in HTML und CSS. Andererseits könnte sich die Erkenntnis ergeben, dass eine gesamtheitliche Betrachtung der drei Sprachen auch hinsichtlich der Code-Qualität dringend notwendig wäre und die beschriebene Forschungslücke dringend geschlossen werden sollte.

Aus den Ergebnissen einer solchen Analyse könnten zusätzlich weitere wertvolle Erkenntnisse zum Umgang mit Code Smells in der Web-Entwicklung gewonnen werden. Aus dem Wissen darüber, welche Smells oder welche Kategorien von Smells besonders stark verbreitet sind, resultiert die Möglichkeit, sinnvolle Priorisierungen vorzunehmen. So kann in der Lehre, der individuellen Fortbildung oder innerhalb von Unternehmen das Bewusstsein von Web-Entwickler*innen gezielt für die häufigsten Code Smells geschärft werden. Dies hätte zur Folge, dass bei geringem Aufwand bereits die Existenz eines großen Anteils von Code Smells verhindert werden könnte.

Zusätzlich könnten die Erkenntnisse aus einer solchen Analyse für die zukünftige Entwicklung von Tools zur automatisierten Erkennung von Code Smells genutzt werden. Welche Smells sollten unbedingt von solchen Tools abgedeckt werden? Bei welchen Smells würde sich der Aufwand einer Optimierung der Erkennung für eine möglichst fehlerfreie Identifikation lohnen?

Diese Arbeit widmet sich der Durchführung einer solchen Analyse und der Gewinnung der genannten Erkenntnisse. Mit *WebDeo* soll aus der Arbeit ein Tool für Web-Entwickler*innen resultieren, welches eine hohe Anzahl verschiedener Code Smells in den Sprachen HTML, CSS und JavaScript auf Websites identifizieren kann. *WebDeo* könn-

te somit zur Verbesserung der Code-Qualität auf Websites beitragen und soll Entwickler*innen zusätzlich über individuelle Stärken und Schwächen hinsichtlich der Qualität des von ihnen geschriebenen Codes aufklären können.

1.2. Ziele

Es gibt bereits einige Arbeiten, welche sich der Untersuchung der Entstehung, der Langlebigkeit und der negativen Auswirkungen der verschiedenen Code Smells widmen. Einige dieser Studien werden im Rahmen von Kapitel 3 vorgestellt. Ein weiterer Aspekt, um die Relevanz der verschiedenen Code Smells ganzheitlich betrachten zu können, ist deren jeweilige Verbreitung. Das Gewinnen von diesbezüglichem Wissen ermöglicht es, Entwickler*innen priorisiert über die häufigsten Code Smells aufzuklären. Dadurch kann mit vergleichsweise geringem Aufwand ein Bewusstsein für und besserer Umgang mit einem großen Anteil der Gesamtheit aller Code Smells geschaffen werden. Insbesondere in Hinblick auf die Web-Entwicklung fehlt es an aussagekräftigen Studien zur Verbreitung der einzelnen Code Smells [BOM20].

Übergeordnetes Ziel dieser Arbeit ist daher das Zusammentragen und Auswerten eines umfangreichen und aussagekräftigen Datensatzes zu der Präsenz von Code Smells in der Web-Entwicklung. Dabei sollen sowohl die Verbreitung von Code Smells in ihrer Gesamtheit als auch die individuelle Häufigkeit der einzelnen Smells untersucht werden. Um das Erreichen dieses Ziels zu ermöglichen, werden folgende Forschungsfragen (RQs) untersucht.

I) Welche Code Smells sind für die Web-Entwicklung relevant?
   a) Welche klassischen Code Smells lassen sich auf die webspezifischen Sprachen HTML, CSS und JavaScript übertragen?
   b) Welche weiteren Code Smells existieren spezifisch für die Strukturierungssprache HTML und die Designsprache CSS?
   c) Gibt es weitere Code Smells im Web, die auf der Kombination der drei Sprachen basieren?

II) Wie häufig treten Code Smells im Web insgesamt auf? Gibt es deutliche Unterschiede zwischen den untersuchten Sprachen?

III) Wie stark sind die einzelnen Code Smells verbreitet?

IV) Weisen Code Smells mit bestimmten Eigenschaften eine höhere Verbreitung auf als andere?

RQ.I) lässt sich durch die Betrachtung der Teilfragen RQ.Ia), RQ.Ib) und RQ.Ic) beantworten. Ihre Untersuchung ist durch die Kombination der Recherche in bestehender Literatur mit eigenständigen begründeten Einschätzungen geplant. Als Ergebnis ergibt

sich eine Liste von relevanten Code Smells im Web, welche die Grundlage für die Beantwortung der Fragen RQ.II), RQ.III) und RQ.IV) darstellt. Auf Basis der Literaturrecherche sollen zusätzlich unterscheidbare Eigenschaften identifiziert werden, die den Code Smells zugeordnet werden können. Die resultierenden Smell-Gruppen mit den jeweiligen Eigenschaften werden genutzt, um Muster im Hinblick auf Frage RQ.IV) zu identifizieren.

Zur Beantwortung von RQ.II), RQ.III) und RQ.IV) soll eine umfassende Untersuchung zufällig ausgewählter Web Pages vorgenommen werden. Dafür soll auf Basis der Rechercheergebnisse ein Tool zur Erkennung von Code Smells im Web entwickelt werden. Das geplante Tool soll über die Angabe einer URL (Uniform Resource Locator) die Dateien der Sprachen HTML, CSS und JavaScript von der zugehörigen Web Page extrahieren. Anschließend sollen die Dateien auf das Vorhandensein von Code Smells untersucht und die Ergebnisse in strukturierter Form abgespeichert werden.

Um aussagekräftige Ergebnisse bezüglich RQ.II), RQ.III) und RQ.IV) zu erhalten, ist die Untersuchung von mindestens 10.000 unterschiedlichen Websites anhand je einer Web Page mithilfe des Analyse-Tools geplant. Gemäß des „Gesetzes der großen Zahlen“ trägt diese umfassende Stichprobe dazu bei, möglichst repräsentative Antworten auf die Forschungsfragen zu erhalten. Die Wahl von 10.000 als sinnvolle Stichprobengröße basiert auf bestehenden Arbeiten mit vergleichbaren Experimenten [KA18, SV21] sowie eigenen Erfahrungen aus einer vorherigen Studie [Sch21]. Die URLs sollen mithilfe eines Web Crawlers zusammengetragen werden, welcher unabhängig von dem Analyse-Tool entwickelt wird.

Eine große Hürde bezüglich des richtigen Umgangs mit Code Smells ist das fehlende Bewusstsein von Entwickler*innen für die Thematik [YM13]. Bestehende Studien zeigen außerdem, dass die manuelle Identifizierung von Code Smells als zu zeitintensiv und fehleranfällig gilt [RA17]. Das Bereitstellen automatisierter Tools zur Erkennung von Code Smells ist somit eine logische Konsequenz zum Angehen der Problematik. Zusätzlich ist der konkrete Wunsch von Entwickler*innen nach besserem Tool-Support in Studien belegt [YM13]. Daher stellt die Weiterentwicklung des Analyse-Tools ein weiteres Ziel dieser Arbeit dar.

Durch das Bereitstellen eines Feedback-Reports soll das Tool um die Möglichkeit ergänzt werden, Web-Entwickler*innen auf das Vorhandensein von Smells in dem Code ihrer Website aufmerksam zu machen. Durch die Angabe empfohlener Refactoringschritte sollen die Nutzer*innen des Tools zusätzlich bei der Überarbeitung des betroffenen Codes unterstützt werden.

Eine weitere Besonderheit des geplanten Tools läge darin, dass die Websites auf Code Smells in verschiedenen Sprachen untersucht werden sollen. Bestehende Tools beschränken sich größtenteils auf eine spezifische Sprache und erkennen relativ wenige Code Smells (⇒ Kapitel 3.2).

Zusätzlich soll die Code-Qualität der analysierten Websites mit den Ergebnissen auf

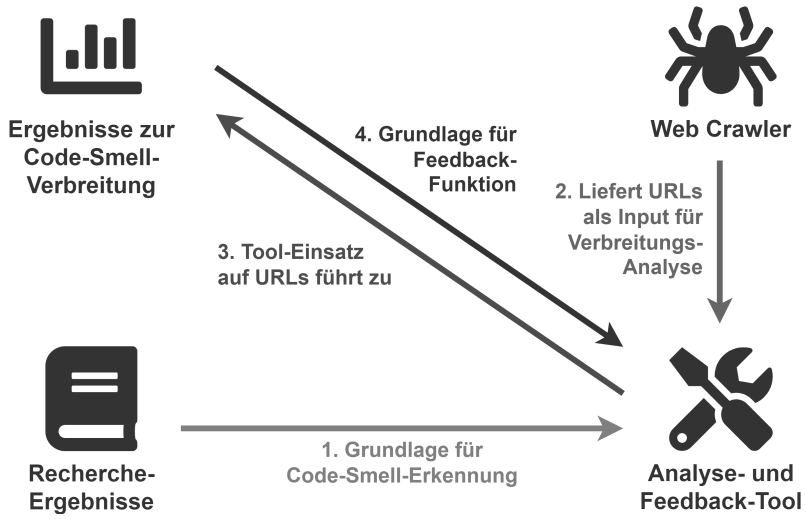


Abbildung 1.1.: Zusammenspiel der geplanten Arbeitsschritte und Zwischenergebnisse

RQ.II) und RQ.III) abgeglichen werden. Somit sollen Stärken und Schwächen der Entwickler*innen im Vergleich zur allgemeinen Code-Qualität im Web festgestellt werden. Die Nutzer*innen des Tools sollen damit die Möglichkeit erhalten, ihr individuelles Verbesserungspotenzial aufgezeigt zu bekommen und nutzen zu können.

Die geplanten Arbeitsschritte und Zwischenergebnisse bauen aufeinander auf und unterstützen sich gegenseitig (⇒ Abbildung 1.1). So sind etwa die Erkenntnisse zur Verbreitung der Code Smells das Resultat des Einsatzes des Analyse-Tools. Gleichzeitig stellen sie aber auch die Grundlage für die anschließende Implementation der Feedback-Funktion des Tools dar.

1.3. Themenspektrum

Zum Erreichen der beschriebenen Ziele werden verschiedene thematische Bereiche der Software-Entwicklung und Informatik miteinander kombiniert. Mit den Themen Code Smells und Refactoring bewegt sich die theoretische Grundlage in den Bereichen der Software-Wartung und der Softwarequalität.

Praktische Anwendung erhält diese Domäne durch das Gebiet der Web-Entwicklung. Mit HTML, CSS und JavaScript wird der Fokus dabei auf die browserseitig interpretierten Web-Sprachen gelegt.

Mit der automatisierten Erkennung von Code Smells erhält der Bereich der statischen Code-Analyse Einzug in das Themenspektrum der Arbeit. Dabei werden Quellcode-Modelle wie der Abstract Syntax Tree (AST) genutzt, um die Analyse von zahlreichen Code-Metriken zu ermöglichen.

Um die Datensammlung für das geplante Experiment zu ermöglichen, wird das Themenspektrum um das Gebiet des Web Minings erweitert. Mit der Sammlung von URLs und der Extraktion von Informationen aus dem Quellcode sind sowohl die Teilgebiete

des Web Crawlings als auch des Web Scrapings vertreten.

2. Theoretische Grundlagen

2.1. Code Smells und Refactoring

Bei Code Smells handelt es sich um suboptimale Design-Entscheidungen auf Code-Ebene. Das Vorhandensein von Code Smells wirkt sich negativ auf die Qualität, Verständlichkeit und Wartbarkeit von Software aus. In der Literatur lassen sich übereinstimmend fünf Hauptmerkmale für Code Smells identifizieren [SS18].

1. Code Smells sind Indikator oder Symptom von weiterreichenden Design-Problemen.
2. Bei Code Smells handelt es sich um suboptimale oder schlechte Lösungen für die gewünschte Funktionalität.
3. Code Smells verstoßen gegen die „Best Practice"-Lösungen der jeweiligen Domäne.
4. Code Smells verschlechtern die Qualität von Software-Systemen und haben somit negativen Einfluss auf deren Weiterentwicklung und Wartung.
5. Bei Code Smells handelt es sich um charakteristische, wiederkehrende Probleme.

Code Smells entstehen vermehrt bei der Entwicklung neuer oder der Erweiterung bestehender Features [TPB+15]. Zeitlich häuft sich ihre Einführung in ein Software-Projekt zu dessen Beginn sowie in besonders arbeitslastigen und stressigen Phasen, etwa vor Deadlines [RA17, TPB+15]. Die Ursachen für die vermehrte Entstehung von Code Smells liegen vor allem im Projektumfeld. Einige Beispiele dafür sind Wissenslücken, mangelnde Erfahrung, häufige Anforderungsänderungen, schlechte Planung, Zeitdruck, Arbeitskultur und -politik sowie das Priorisieren von Quantität über Qualität [SS18, RA17]. Zusätzlich können technische Einschränkungen durch die verwendete Sprache, Technologie oder Plattform die Einführung von Code Smells fördern [SS18].

Code Smells unterscheiden sich von Bugs, denn sie sorgen nicht dafür, dass ein unerwartetes Verhalten beim Ausführen der Software eintritt [AC20, Ble18]. Gleichzeitig erhöht das Vorhandensein von Code Smells langfristig das Risiko des Auftretens von Fehlern [JKA19, YM12]. Ursächlich dafür ist die negative Auswirkung auf die Verständlichkeit und langfristige Wartbarkeit der Software [TPB+15, JKA19, YM12]. Werden Änderungen vorgenommen, ohne dass die Funktionsweise des Codes korrekt verstanden wird, werden Fehler und Abweichungen von der gewünschten Funktionalität in die Software eingebracht [Fow20].

Aufgrund der Verschlechterung der Wartbarkeit können Code Smells langfristig viele weitere Probleme verursachen. Sie können negative Auswirkungen auf die Zuverlässig-

keit, Testbarkeit und Performance der Software haben [SS18]. Zusätzlich sorgt das Vorhandensein von Code Smells dafür, dass die betroffenen Code-Einheiten deutlich häufiger als üblich verändert werden müssen [KDPG09, SS18]. Eine auffallend starke Verbreitung von Code Smells resultiert zusätzlich in einem zermürbenden Effekt auf die Entwickler*innen und schadet deren Arbeitsmoral [SS18].

Code Smells sind grundlegender Bestandteil der „technischen Schulden“ innerhalb eines Projektes [TPB+15]. Diese Metapher beschreibt die negativen Auswirkungen (⇒ Schulden) von schlechter Software-Entwicklung [TAV13]. Ignoriert man das Vorhandensein von Code Smells, treten langfristig die zuvor beschriebenen Konsequenzen ein, welche die Weiterentwicklung der Software massiv behindern können [Fow20, SS18]. Durch den gesteigerten Aufwand der Änderungen steigen auch die Änderungskosten. Code Smells tragen somit – im wahrsten Sinne des Wortes – zum Aufbau technisch bedingter Schulden bei.

Martin Fowler fasst die Thematik einer hohen Code-Qualität, welche durch Code Smells reduziert werden würde, treffend zusammen:

> *„Wenn jemand eine Änderung vornehmen muss, sollte der zu ändernde Code schnell und einfach zu finden und zu ändern sein, ohne dass dies irgendwelche Fehler nach sich zieht. Eine gute Codebasis maximiert unsere Produktivität und ermöglicht es uns so, zusätzliche Features für unsere Anwender sowohl schneller als auch kostengünstiger zu entwickeln.“*
>
> — Martin Fowler *[Fow20]*

Um die negativen Folgen von Code Smells zu verhindern, sollte deren Einführung also möglichst vermieden werden. Die initiale Entwicklung von Features gilt jedoch als sehr komplex – häufig fokussieren sich Entwickler*innen eher auf das Erreichen der gewünschten Funktionalität und weniger auf die Code-Qualität [Fow20]. Daher kommt der Identifikation und anschließenden Entfernung bestehender Code Smells hohe Relevanz zu. Das Auffinden von Code Smells durch Entwickler*innen ist stark von deren Fähigkeiten abhängig und gilt als zeit- und kostenaufwändige Aktivität [RA17, FM13]. Daher ist der Bedarf für automatisierte Tools zur Identifizierung von Code Smells hoch [YM13]. Bezüglich der Entfernung identifizierter Code Smells gibt es hingegen sehr konkrete, äußerst verständliche und gut erforschte Vorgehensweisen: „Refactorings“.

Refactorings sind kleinschrittige Überarbeitungsmaßnahmen auf Code-Ebene, die die Qualität und Struktur des Codes verbessern sollen, ohne dabei die Funktionalität zu verändern [Fow20]. Durch den Prozess des Refactorings können Code Smells gezielt entfernt werden. Es resultiert „saubere[r] Code, der einfacher gewartet, einfacher debuggt und einfacher um neue Features ergänzt werden kann“ [Har12].

Code Smells und Refactorings gehen dabei eine Art Symbiose ein. Während Code Smells ein Indikator dafür sind, an welchen Stellen Refactoring angewendet werden sollte [SS18], sind für die verschiedenen Smells konkrete Refactorings für deren Beseitigung

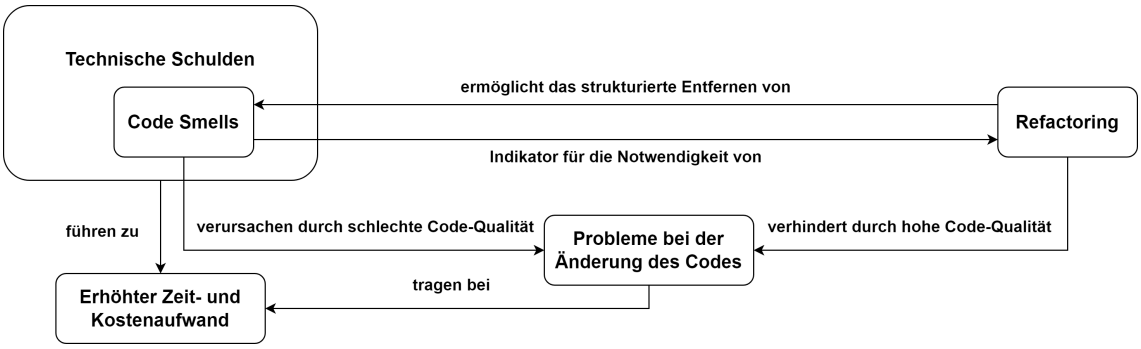


Abbildung 2.1.: Zusammenhang von Code Smells, Refactoring und Technischen Schulden

beschrieben [Fow20]. Somit eignen sich Code Smells einerseits als Indikator zum Erzielen der Vorteile des Refactorings – und gleichzeitig eignen sich konkrete Refactorings optimal zum Entfernen von Code Smells und ihrer negativen Effekte.

Das wichtige Zusammenspiel der beschriebenen Themen (⇒ Abbildung 2.1) lässt sich wie folgt zusammenfassen. Code Smells sind Indikatoren für Qualitätsschwächen im Code sowie für die Notwendigkeit von Refactoring. Werden Code Smells nicht entfernt, drohen langfristig Probleme bei der Änderung des Codes. Durch die Anwendung konkreter Refactoringschritte können Code Smells entfernt und ihr Beitrag zu der Gesamtheit an technischen Schulden minimiert werden. Es entsteht gut verständlicher Code, welcher sich zeit- und kosteneffektiver warten und erweitern lässt.

2.2. Automatisierte Erkennung von Code Smells

2.2.1. Statische Code-Analyse

Es gibt verschiedene Ansätze zur automatisierten Erkennung von Code Smells. Für die geplante Analyse und das zu entwickelnde Tool wird sich mit der statischen Code-Analyse auf die am weitesten verbreitete Methode fokussiert. Weitere Strategien zur automatisierten Erkennung von Code Smells werden in Kapitel 2.2.3 kurz vorgestellt.

Ermöglicht wird das statische Analysieren des Codes mithilfe von Konzepten, die auch beim Kompilieren genutzt werden, allen voran der Nutzung des Abstract Syntax Trees [FM13, Sab16, AL20]. Der AST ist das Resultat des Parsens der grundlegenden Struktur des Quellcodes in eine baumartige Repräsentationsform, welche automatisiert analysiert werden kann [Sab16]. Die verschiedenen im Quellcode genutzten Elemente, wie zum Beispiel Schleifen, Bedingungen, Wertzuweisungen oder Return-Statements, stellen dabei die Knoten der Baumstruktur dar. Zur weiteren Veranschaulichung zeigt Abbildung 2.2 einen Ausschnitt eines ASTs im JSON-Format (links) und eine grafische Repräsentation des selbigen (rechts). Der dargestellte AST bezieht sich auf einen Teil des Codes, welcher in Abbildung 2.6 zu sehen ist. Neben dem AST können auch andere Quellcode-Modelle die statische Analyse des Codes ermöglichen [SS18].

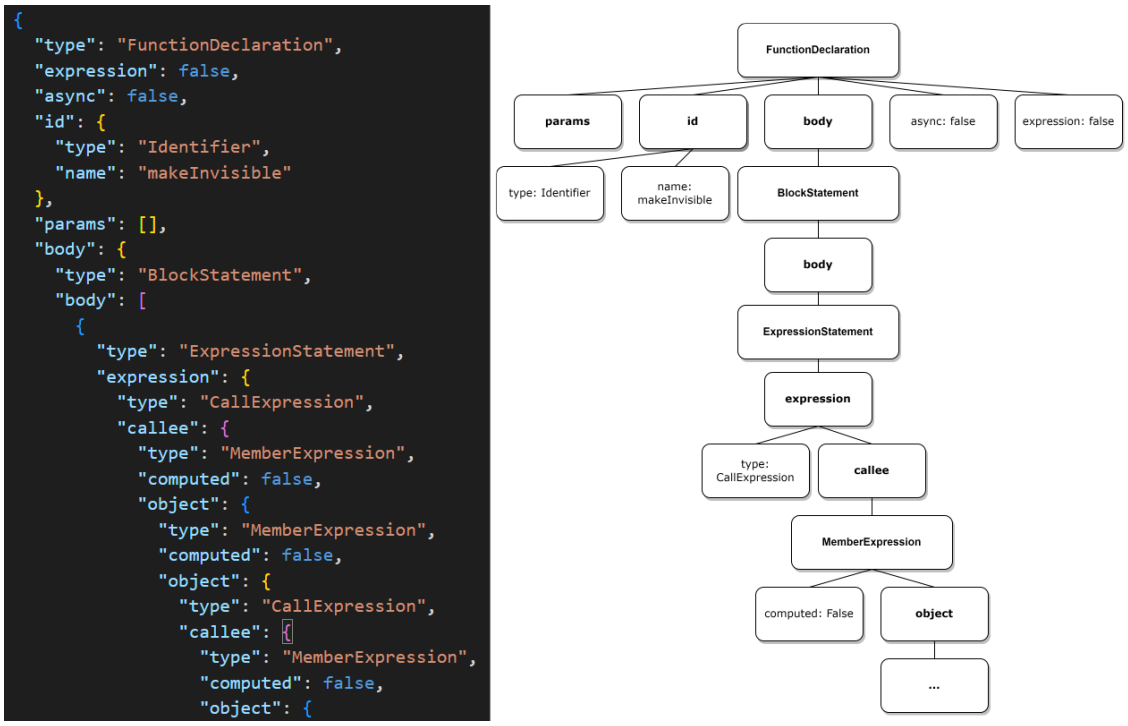


Abbildung 2.2.: Ausschnitt eines beispielhaften ASTs im JSON-Format (links) und grafisch repräsentiert (rechts)

Über die Möglichkeiten der statischen Code-Analyse werden für den untersuchten Code Kennwerte ermittelt, welche für Code Smells entscheidend sind. Neben dem AST bietet die Nutzung weiterer Technologien, wie regulärer Ausdrücke oder die Analyse von Strings, dabei Unterstützung [RA17]. Ob ein Code Smell vorliegt, wird durch den Vergleich mit festgelegten Schwellenwerten entschieden [SS18]. Das Konzept dieser Kennwerte und Schwellenwerte zur Erkennung von Code Smells wird in Kapitel 2.2.2 näher betrachtet.

Die statische Code-Analyse eignet sich vor allem zum Erkennen von Code Smells, welche keine tiefgreifenden Ursachen besitzen [SMK17, RA15]. So ist die Erkennung suboptimaler Lösungen innerhalb kleiner Programmeinheiten, beispielsweise Parameterlisten oder Fallunterscheidungen, sehr gut möglich. Code Smells, welche zum Beispiel Brüche mit klassenübergreifenden Regeln der Objektorientierung darstellen, könnten hingegen kaum von diesem Ansatz erkannt werden.

Für die im Rahmen dieser Arbeit geplanten Analyse und Entwicklung fällt die Wahl einer Erkennungsstrategie auf die statische Code-Analyse. Ausschlaggebend für diese Entscheidung ist, dass dieser Ansatz die besten Aussichten zur Übertragung von Programmiersprachen auf die Strukturierungssprache HTML und die Designsprache CSS bietet. Für beide Sprachen existieren mit dem AST vergleichbare Quellcode-Modelle [Ric, Sap]. Zusätzlich setzt die dritte analysierte Sprache, JavaScript, Strukturen der Objektorientierung nicht im klassischen Sinne um [SMK17]. Folglich ergibt es keinen Sinn, sich für das geplante Vorhaben auf Code Smells zu fokussieren, welche zum Beispiel komplexe

Zusammenhänge zwischen Klassen oder Vererbungsstrukutren thematisieren, da diese Elemente in vielen JavaScript-Projekten keine Anwendung finden. Stattdessen wird der Fokus auf weniger tiefgreifende Code Smells gelegt, welche außerdem deutlich besser durch die Mittel der statischen Code-Analyse identifiziert werden können.

2.2.2. Metriken und Schwellenwerte

Die Kennwerte zur Identifikation von Code Smells werden als „Metriken“ bezeichnet. Metriken repräsentieren Eigenschaften des Codes als Wahrheitswert (Datentyp `boolean`) oder in ganzzahliger Form (Datentyp `int`). Der Wert boolescher Metriken (*wahr* oder *falsch*) kann über das Stellen einfacher Fragen ermittelt werden. Ein Beispiel hierfür ist die Frage „*Wird in einem Bedingungsstatement ein Zuweisungsoperator anstelle eines Vergleichsoperators genutzt?*“ [JKA19]. Die Anzahl an Zeilen in einer Methode oder von Parametern in der Methodendeklaration sind exemplarische Integer-Metriken [FM13].

Zum Fällen der Entscheidung, ob ein Code Smell vorliegt, werden die ermittelten Werte des Codes für Integer-Metriken mit zuvor definierten Schwellenwerten verglichen. Wird der gesetzte Schwellenwert abhängig von dessen Bedeutung über- oder unterschritten, liegt ein Code Smell vor [JKA19]. Der Wahl der richtigen Schwellenwerte kommt daher eine große Bedeutung zu – gleichzeitig gibt es aber kaum wissenschaftlichen Konsens bezüglich der Definition von Standardwerten [RA15].

2.2.3. Weitere Erkennungsstrategien

Neben den Möglichkeiten der statischen Code-Analyse gibt es weitere Ansätze zur Erkennung von Code Smells. Über die verschiedenen Erkennungsstrategien lassen sich auch unterschiedliche Code Smells erkennen.

Mit der Einführung von Code Smells durch Fowler etablierten sich „manuelle Techniken“ zur Erkennung von Code Smells durch Entwickler*innen [RA15]. Durch die Anwendung von bestimmten Inspektionstechniken wird der Code mit dem Fokus auf Verstöße gegen Designprinzipien analysiert [KDPG09]. Manuelle Techniken zur Code-Smell-Erkennung gelten als zeitintensiv und fehleranfällig; somit eignen sie sich nicht für die Anwendung auf große Systeme [RA15].

„Regelbasierte Techniken“ zur Code-Smell-Erkennung überprüfen ein Modell der Software auf Verstöße gegen Designregeln. Mit diesem Ansatz können zum Beispiel klassenübergreifende Smells, wie das Vorhandensein von zyklischen Klassen-Abhängigkeiten, identifiziert werden [SS18].

Ein weiterer Ansatz zum Erkennen von Code Smells liegt im Bereich des „Machine Learnings“. Durch die Analyse von zahlreichen Software-Projekten und Designregeln wird ein Modell so trainiert, dass es empfehlenswerte Lösungen für typische Herausforderungen auf Code-Ebene erlernt [SS18]. Anschließend lässt sich das Modell auf weitere Projekte anwenden, um Verstöße gegen diese Empfehlungen, also potenzielle Code

Smells, zu identifizieren. Der Erfolg dieser Ansätze hängt sehr von der Datenqualität und dem Trainingsprozess ab [RA15].

Mit Hinblick auf die Besonderheiten der Sprache JavaScript gibt es zusätzlich den Ansatz der „dynamischen Analyse“. Dabei wird die Erstellung und Aktualisierung von Objekten, Funktionen und Variablen zur Laufzeit durch direkte Interaktion des Erkennungstools mit dem Browser untersucht [FM13]. Da das Programm für diesen Ansatz ausgeführt werden muss, gilt er als sehr zeitintensiv [SMK17] und eignet sich daher auch nicht für die im Rahmen dieser Arbeit geplanten Analyse mit dem Ziel einer großen Stichprobe. Die dynamische Analyse erlaubt die Erkennung des Zusammenspiels verschiedener Code-Komponenten, welche in JavaScript nicht auf Basis statischer Code-Analyse stattfinden kann [FM13]. Da der Programmablauf von JavaScript stark von der Interaktion der Nutzer*innen mit Buttons und ähnlichen HTML-Elementen abhängt, bleiben viele Szenarien dennoch unentdeckt [SMK17].

Außerdem gibt es weitere Strategien zum Erkennen von Code Smells, welche allerdings weniger verbreitet sind. Beispiele hierfür sind symptom- [RA15], historien- [SS18], SQL- [RA17] und optimisierungsbasierte [SS18] Ansätze. Zusätzlich werden Mittel aus dem Bereich der Visualisierung genutzt, um bestimmte Metriken, und somit potenzielle Code Smells, optisch darzustellen [RA15, KDPG09].

2.3. Sprachen der Web-Entwicklung

2.3.1. HTML

Die Hypertext Markup Language (HTML) ist für die Strukturierung von Elementen auf einer Web Page zuständig. Ruft ein Browser eine Web Page auf, wird er exakt ein HTML-Dokument auffinden, aus dem heraus unterschiedlich viele CSS- oder JavaScript-Dateien eingebunden werden können. HTML ist somit der unverzichtbare Kern einer jeden Web Page.

HTML ist von zwei grundlegenden Elementen geprägt. Eine Auswahl an „Tags“ vermittelt den Browsern, wie die verschiedenen Elemente auf der Web Page zu interpretieren sind [Rob12]. So gibt es beispielsweise Tags für die Definition von Textparagraphen, Überschriften, Hyperlinks, Bildern und Videos. Durch das Verschachteln und Aneinanderketten der Tags entsteht die vollständige Struktur der Web Page, welche als Document Object Model (DOM) in einer baumartigen Struktur dargestellt wird. „Attribute“ ermöglichen die Zuordnung weiterer Eigenschaften zu einzelnen Tags [TN11].

Abbildung 2.3 zeigt beispielhaft das Zusammenspiel der unterschiedlichen HTML-Elemente. Das `img`-Tag wird mithilfe des `src`-Attributs um einen Dateipfad für das Bild erweitert. Auf das Bild folgen einige aneinandergereihte Textparagraphen (`<p>`). In dem mittleren Paragraph ist wiederum ein Hyperlink (`<a>`) geschachtelt. Über das `id`-Attribut können einzelne Tags benannt werden, mithilfe des `class`-Attributes können ähnliche Elemente in Klassen zusammengefasst werden. Sowohl über Identifiers (IDs) als

```
<img src="hund.jpg" id="bild-hund" />
<p class="wichtig">
    Auf der gesamten Welt leben etwa 500.000.000 Hunde, welche als Haustiere gehalten werden.
</p>
<p>
    Der Hund stammt von dem <a href="https://de.wikipedia.org/wiki/Wolf">Wolf</a> ab.
</p>
<p class="wichtig">
    Einige Sinne des Hundes gelten als hochentwickelt. So können Hunde etwa deutlich besser
    hören und riechen als Menschen.
</p>
```

Auf der gesamten Welt leben etwa 500.000.000 Hunde, welche als Haustiere gehalten werden.

Der Hund stammt von dem Wolf ab.

Einige Sinne des Hundes gelten als hochentwickelt. So können Hunde etwa deutlich besser hören und riechen als Menschen.

Abbildung 2.3.: Beispielhafter HTML-Code (oben) und resultierende Web Page (unten)

auch über Klassen können HTML-Elemente aus CSS und JavaScript heraus referenziert werden. Die resultierende Web Page schachtelt und reiht die Elemente wie beschrieben aneinander und erkennt bereits ihre Funktion. Die IDs und Klassen sorgen derzeit jedoch noch für keinen sichtbaren Unterschied.

2.3.2. CSS

Die in Abbildung 2.3 dargestellte Web Page ist optisch noch nicht sehr ansprechend. An diesem Punkt kommt mit Cascading Style Sheets (CSS) die zweite elementare Sprache des Webs ins Spiel. CSS besitzt eine simple Syntax und erlaubt das Setzen konkreter Werte für verschiedene Design-Eigenschaften von HTML-Elementen.

Durch die bewusste und vollständige Separation von CSS als Designkomponente und HTML als Strukturierungskomponente wurde eine klare Zuständigkeitstrennung in der Web-Entwicklung erzielt. Zusätzlich resultierte daraus die Möglichkeit zur unabhängigen Arbeit von Designern und Entwicklern und die höchstrelevante Möglichkeit der Wiederverwendung von CSS-Dateien [Ble18]. Somit konnte unterschiedlichen Web Pages derselben Website ein einheitliches Design zugeordnet werden, ohne die Nachteile von Code-Wiederholung akzeptieren zu müssen. Während frühere HTML-Versionen sogar Tags mit Designfunktionalität enthielten, wird heute gänzlich davon abgeraten, CSS in einer anderen Form als über separate Dateien in HTML einzubinden [NNN+12].

Abbildung 2.4 zeigt die Veränderung der Web Page im Vergleich zu Abbildung 2.3 durch das Hinzufügen weniger „CSS-Regeln“. Regeln enthalten Deklarationsabschnit-

```css
#bild-hund {
    width: 80%;
    margin: 10px 10%;
}

p {
    width: 100%;
    text-align: justify;
    font-family: Helvetica;
}

.wichtig {
    font-weight: bold;
}

p a {
    color: darkgreen;
}

html {
    background-color: lightblue;
}
```

Auf der gesamten Welt leben etwa 500.000.000 Hunde, welche als Haustiere gehalten werden.

Der Hund stammt von dem Wolf ab.

Einige Sinne des Hundes gelten als hochentwickelt. So können Hunde etwa deutlich besser hören und riechen als Menschen.

Abbildung 2.4.: Beispielhafter CSS-Code (links) und resultierende Website (rechts)

te, welche sich durch geschweifte Klammern öffnen und schließen lassen (⇒ Abbildung 2.5). Einzelne Deklarationen bestehen aus der Angabe eines „Attributs“ und einem oder mehrerer „Werte“ für ebendieses. So sorgt in diesem Beispiel die Deklaration `color: darkgreen;` dafür, dass das Schriftfarbe-Attribut die Farbe Dunkelgrün als Wert zugewiesen bekommt.

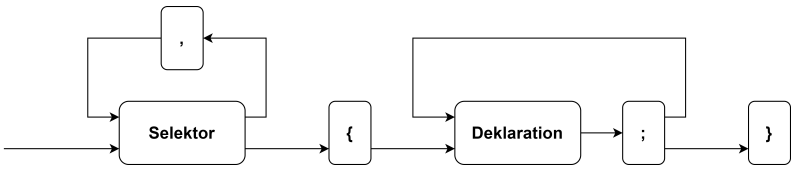


Abbildung 2.5.: Darstellung der grundlegenden CSS-Syntax
Eigene Darstellung in Anlehnung an Gharachorlu [Gha14]

Regeln gelten für alle HTML-Elemente, die durch die „Selektoren“, welche vor den geschweiften Klammern stehen und durch Kommata voneinander getrennt werden, beschrieben werden (⇒ Abbildung 2.5). Selektoren lassen sich vielseitig nutzen. So können zum Beispiel alle Tags einer Sorte (`p`), Elemente mit einer bestimmten ID (`#bild-hund`) oder alle Elemente einer Klasse (`.wichtig`) angesprochen werden. Die verschiedenen Selektorenelemente lassen sich auch miteinander kombinieren. So würde die Angabe (`p.wichtig`) nur Textparagraphen ansprechen, welche zusätzlich die Klasse `wichtig` zugewiesen bekommen haben. Durch das Hintereinandersetzen verschiedener Selektoren lässt sich auch gezielt die Verschachtelung von HTML-Elementen ansprechen. Der Selektor `p a` in Abbildung 2.4 bezieht sich beispielsweise ausschließlich auf Hyperlinks innerhalb von Textparagraphen.

2.3.3. JavaScript

Die dritte Sprachkomponente, welche aufwändige Web-Anwendungen erst ermöglicht, ist JavaScript. JavaScript übernimmt als Programmiersprache die funktionalen Aufgaben einer Website. Grundlegende Charakteristiken von JavaScript sind die dynamische Typisierung und das Unterstützen von objektorientieren Konzepten ohne die klassische Umsetzung von Klassen [FM13]. Zusätzlich wird JavaScript browserseitig interpretiert und der Quellcode vorab nicht kompiliert oder geprüft. Diese Eigenschaften haben zur Folge, dass JavaScript als besonders fehleranfällig und vergleichsweise komplex in der Entwicklung gilt [JKA19]. Häufig wird JavaScript daher in Verbindung mit Frameworks genutzt, die die Entwicklung vereinfachen sollen.

JavaScript ist in der Lage, zur Laufzeit mit HTML und CSS zu interagieren und so zum Beispiel dynamische Änderungen an dem DOM vorzunehmen [FM13]. Somit ist die fortlaufende Veränderung der Inhalte und des Aussehens einer Web Page – nachdem sie initial geladen ist – möglich. Ein weiterer Durchbruch für die Web-Entwicklung war die Einführung des Konzeptes „Asynchronous JavaScript And XML“ (AJAX). Es ermöglichte den Datenaustausch mit Servern aus JavaScript heraus – ebenfalls ohne die Notwendigkeit, die Web Page neu zu laden [w3sa].

JavaScript kann wie CSS über verschiedene Wege in HTML-Code integriert werden. Um die klare Trennung von Verantwortlichkeiten zu unterstützen, gilt jedoch ebenfalls nur die Einbindung externer JavaScript-Dateien als empfehlenswert [NNN+12, AC20].

```
var imageID = "bild-hund";
var imageObject = document.getElementById(imageID);
imageObject.addEventListener("click", makeInvisible);

function makeInvisible() {
    document.getElementById(imageID).classList.add("invisible");
}
```

Abbildung 2.6.: JavaScript-Code zum Ausblenden des Bildes auf der Beispiel-Website

Abbildung 2.6 zeigt eine simple JavaScript-Lösung, die das Bild auf der beispielhaften Web Page ausblendet, wenn es angeklickt wird. Dafür wird das Bild-HTML-Element `#bild-hund` aus dem DOM heraus angesprochen und mit einem Event-Listener versehen. Wird der Event-Listener nun durch einen Klick auf das Bild aktiviert, ruft er die Methode `makeInvisible` auf. Sie fügt die Klasse `invisible` zum Bild-Element hinzu, wodurch es unsichtbar wird. Im CSS-Code muss dafür die Regel `.invisible {display: none;}` existieren.

2.4. Bedeutung von Code Smells in der Web-Entwicklung

Die Wichtigkeit und das Potenzial der Themen Code Smells und Refactoring in der Web-Entwicklung sind in der Literatur umfassend belegt. Insbesondere für HTML und CSS

ergeben sich allerdings Besonderheiten hinsichtlich der Übertragung dieser Themen, da es sich bei ihnen nicht um Programmiersprachen handelt.

Im Vergleich zu CSS und insbesondere JavaScript sind Code Smells bislang vergleichsweise wenig auf HTML übertragen worden. Anders sieht es im Hinblick auf Refactoring aus. Wenngleich die Refactoring-Schritte für HTML variieren können, da es sich um eine Strukturierungssprache handelt, spricht Martin Fowler von einer Übertragbarkeit der „Philosophie des Refactorings“ und der Vorteile von sauberem Code auf HTML [Har12].

Auch die grundlegenden Eigenschaften von Code Smells lassen sich auf HTML übertragen. Wie auch in Programmiersprachen ist es in HTML möglich, unnötig komplexe Lösungen für das Erzielen einer bestimmten Struktur zu entwickeln. Dies schadet der Verständlichkeit und erschwert die Änderung des Codes. Auch die Eigenschaft, dass Code Smells als charakteristische und wiederkehrende Probleme angesehen werden, kann auf typische ungünstige Vorgehensweisen in der HTML-Entwicklung übertragen werden. Es ist jedoch zu beachten, dass sich zahlreiche bekannte Code Smells für Programmiersprachen nicht auf HTML anwenden lassen, da die Übertragbarkeit für zahlreiche Konzepte, wie beispielsweise Klassen und Methoden, nicht gegeben ist. Gleichzeitig ergeben sich für die Elemente von HTML eigene Code Smells, von denen einige in Kapitel 5.3 vorgestellt werden.

Auch für CSS gibt es bekannte Code Smells, welche sich auf die Designsprache anwenden lassen [PVZ16]. Wie auch bei HTML sind jedoch einige Konzepte aus Programmiersprachen nicht übertragbar – ebenso wie die Code Smells, welche sich darauf beziehen. In Anbetracht der Tatsache, dass CSS keine Programmiersprache ist, sind in der Literatur relativ viele CSS-spezifische Code Smells beschrieben, welche in Kapitel 5.4 näher betrachtet werden.

Trotz der grundlegend simplen Syntax gibt es in CSS komplexe Features, welche dazu beitragen, dass der Code schwierig zu verstehen und zu ändern ist. Zusätzlich gibt es vergleichsweise wenig Tool-Support. Der Prozess der Entwicklung von CSS ist somit grundlegend sehr anfällig für das Einbringen von Code Smells [PVZ16].

Da es sich bei JavaScript um die einzige Programmiersprache der drei betrachteten Web-Technologien handelt, gibt es bereits viele Arbeiten, welche die Bedeutung von Code Smells für diese Domäne untersuchten. Dabei wird übereinstimmend berichtet, dass JavaScript besonders anfällig für das Einbringen von Code Smells ist. Ursächlich dafür sind die vielen Features und die hohe Dynamik der Sprache sowie das Fehlen strenger Einschränkungen und eines Kompiliervorgangs [HLMHC17, SMK17, FM13]. Shoenberger et al. beschreiben JavaScript aufgrund dieser Eigenschaften als „Wartungs-Alptraum von Entwickler*innen“ [SMK17]. Aufgrund des Fehlen eines Compilers, welcher potenzielle Fehler aufdecken könnte, wird die Erkennung von Code Smells für JavaScript gleichzeitig als „präventive Maßnahme zum Minimieren der Anzahl von Bugs“ [SMK17] beschrieben. Somit sind Code Smells in JavaScript überdurchschnittlich bedeutsam; gleichzeitig ist die Sprache besonders anfällig für das Einbringen ebenjener.

Aufgrund der späten und häufig sehr speziellen Umsetzung von objektorientierten Konzepten, wie Klassen und Objekten, sind einige Code Smells aus anderen Programmiersprachen kaum oder gar nicht auf JavaScript anwendbar [FM13, SMK17]. Durch die Besonderheiten der Sprache ergeben sich gleichzeitig jedoch einige spezifische Code Smells für JavaScript, welche in Kapitel 5.5 betrachtet werden.

Abschließend resultieren aus der Kombination der drei Technologien einige spezifische Code Smells für die Web-Entwicklung. Sie basieren auf dem Konzept der „Separation of Concerns“ (SoC, *dt.: „Trennung der Zuständigkeiten“*) [FM13]. Code Smells aus dieser Kategorie umfassen daher verschiedene Möglichkeiten, wie HTML, CSS und JavaScript vermischt werden.

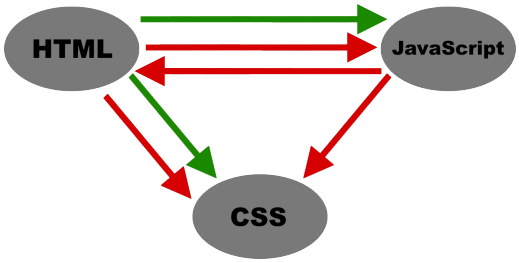


Abbildung 2.7.: Möglichkeiten zur Nutzung einer Sprache innerhalb einer Anderen

Die Möglichkeiten der Sprachkombination werden in Abbildung 2.7 dargestellt. Die grünen Pfeile repräsentieren dabei die gewünschte Einbindung von separaten CSS- und JavaScript-Dateien in HTML. Alle weiteren Kombinationen (durch rote Pfeile dargestellt) gelten als unerwünscht und bilden somit den Bereich von SoC-Smells. Es fällt auf, dass CSS lediglich in HTML und JavaScript genutzt werden kann, jedoch keine der beiden Sprachen in CSS anwendbar ist. Die verschiedenen Code Smells mit Bezug zur Separation of Concerns werden in Kapitel 5.6 vorgestellt.

2.5. Web Crawling

Für das Zusammentragen von den URLs der Web Pages, die auf Code Smells untersucht werden sollen, soll ein „Web Crawler“ entwickelt werden. Web Crawling ist eines der Hauptgebiete des Web Minings und stellt die Grundlage für Suchmaschinen dar. Web Crawler rufen fortlaufend Web Pages auf und extrahieren Hyperlinks, die andere Web Pages und Inhalte referenzieren [GJ09]. Anschließend werden die dadurch aufgefundenen Web Pages ebenfalls auf weitere Verlinkungen untersucht. Somit decken Web Crawler die netzwerkartige Struktur des Webs auf und entdecken neu veröffentlichte Web Pages und Dateien, welche anschließend von Suchmaschinen berücksichtigt werden können [GJ09].

3. Verwandte Arbeiten

3.1. Analysen der Eigenschaften von Code Smells

Verschiedene Aspekte von Code Smells wurden bereits in zahlreichen Arbeiten untersucht. Dabei lassen sich drei grundlegende Themengebiete mit weiteren Teilgebieten identifizieren:

- Das Auftreten von Code Smells in Software-Projekten
  - Entstehung von Code Smells [TPB+15, SSS14, LR15, CM10, MBC14, TAV13]
  - Verbreitung und Häufigkeit von Code Smells [PBP+18, BQO+12, PAN+19, OCAD21, CCMX16]
  - Lebensdauer von Code Smells in Projekten [PZ12, CM10, OCBZ09]
- Die Auswirkungen von Code Smells auf Software
  - Auswirkungen auf die Änderunghäufigkeit der Software [KPGA12, OCBZ09, PBP+18, KDPG09]
  - Verschlechterung der Software-Qualität und daraus resultierende Fehler [KPGA12, JGHK13, SSS14, SL06, AKGA11, TAV13, PBP+18]
- Code Smells und Entwickler*innen
  - Auswirkungen von Code Smells auf die Moral und Produktivität von Entwickler*innen [SSS14, TAV13]
  - Bewertung von Code Smells und ihrer Relevanz durch Entwickler*innen [YM13]

Die referenzierten Analysen beziehen sich größtenteils auf „klassische“ Programmiersprachen der Anwendungsentwicklung, allen voran Java. Die Anzahl an Untersuchungen, die Code Smells thematisieren, wird jedoch deutlich geringer, wenn man den Kontext der Web-Entwicklung mit all seinen Besonderheiten heranzieht. Die Arbeiten, die jedoch innerhalb dieses Themenspektrums existieren, werden im Folgenden mitsamt ihrer grundlegenden Erkenntnisse vorgestellt.

Fard und Mesbah wandten ihr Tool *JSNOSE* an, um 13 verschiedene JavaScript-Code-Smells in elf umfassenden Web-Anwendungen aufzuspüren [FM13]. Die Ergebnisse besitzen aufgrund der geringen Stichprobe eine begrenzte Aussagekraft und sprechen eher für die spezifischen Web-Anwendungen als für die allgemeine Verbreitung der verschiedenen Code Smells. Am häufigsten wurden die Code Smells *Lazy Object*, *Long Method*, *Excessive Global Variables*, *SoC*-Smells sowie eine Gruppe von Smells, welche sich auf das

„Closure"-Feature in JavaScript beziehen, identifiziert. Diese Code Smells traten jeweils in mindestens 64% der untersuchten Web-Apps auf.

Agrahari und Chimalakonda nutzten ebenfalls *JSNOSE*, um 13 Code Smells in 361 JavaScript-Web-Games zu identifizieren [AC20]. In diesem spezifischen Kontext traten die Smells *Lazy Element* und *Chained Methods*, mit einem relativen Anteil von 36,38% und 28,22% an der Gesamtheit aller identifizierten Code Smells, besonders häufig auf.

Saboury et al. stellten eine Untersuchung dazu an, ob JavaScript-Dateien, die Code Smells enthalten, in der weiteren Entwicklung fehleranfälliger sind als solche ohne Smells [SMKA17]. Zusätzlich wurde untersucht, welche der untersuchten Smells häufiger zu Fehlern führen als andere. Innerhalb von fünf Open-Source-Web-Projekten mit tausenden Commits wurden dafür zwölf JavaScript-Code-Smells identifiziert. Die Analyse zeigte, dass die gleichen Code Smells in unterschiedlichen Projekten auch eine unterschiedliche Fehlerverursachungsrate besitzen. Über alle untersuchten Projekte hinweg wiesen besonders die untersuchten Smells *Variable Re-Assign* und *Assignment in Conditional Statement* eine hohe Fehlerrate auf. Zusätzlich ergab sich die Erkenntnis, dass JavaScript-Dateien ohne Code Smells um 65% weniger fehleranfällig sind als solche mit Code Smells.

Die Analyse von Saboury et al. wurde 2019 von Johannes et al. mit erweitertem Untersuchungsgegenstand und einer größeren Stichprobe von 15 Web-Projekten wiederholt [JKA19]. In dieser Untersuchung wurden Fehler, die auf das Vorhandensein von Code Smells folgten, nicht mehr auf dem Level von Dateien, sondern spezifisch auf die Code-Zeilen um den Smell herum untersucht. Es ergab sich daher mit 45% eine erwartbar etwas niedrigere Rate an durch Code Smells ausgelösten Fehlern als im vorherigen Experiment. Im Hinblick auf die Fehlerverursachungsrate der einzelnen Code Smells ergab sich ein ähnliches Gesamtbild zur vorherigen Analyse. Zusätzlich wurde die Langlebigkeit der einzelnen Code Smells in Projekten über zahlreiche Commits untersucht. Es zeigte sich, dass ein hoher Anteil an Code Smells bereits bei dem Anlegen neuer JavaScript-Dateien entsteht. Zusätzlich wurden mehr als 20% der Code Smells über eine sehr lange Zeit hinweg nicht entfernt. Mit *Variable Re-Assign* wurde ein Code Smell identifiziert, welcher gleichzeitig eine sehr hohe Fehlerverursachungsquote besitzt und durchschnittlich am längsten von allen untersuchten Smells bestehen bleibt. Die Autor*innen resultieren daher, dass dieser Smell beim Refactoring von JavaScript-Projekten priorisiert behandelt werden sollte.

Bleisch untersuchte, wie hoch der Anteil von CSS-Code-Smells auf Websites ist, die mithilfe unterschiedlicher Frameworks entwickelt wurden [Ble18]. Insgesamt wurden dafür 5.525 Websites, entwickelt mithilfe von 19 verschiedenen Frameworks, auf das Vorhandensein von zwölf CSS-Smells untersucht. Zur Identifizierung der Code Smells wurde das Tool *CSSNose* angewendet. Es zeigte sich eine sehr hohe Verbreitung der verschiedenen Code Smells über alle Frameworks und untersuchten Websites hinweg. Nur selten befanden sich die jeweiligen Code Smells auf weniger als 70% der Websites eines

jeweiligen Frameworks, meistens lag die Quote sogar über 90%.

Dass Code Smells in CSS eine auffallend hohe Verbreitung aufweisen, ist auch die grundlegende Erkenntnis der Arbeit von Gharachorlu [Gha14]. Er untersuchte 500 Websites auf das Vorhandensein von 26 CSS-Smells. Lediglich auf einer einzigen Website (0,2%) wurde keiner der Code Smells nachgewiesen. Auf jeder zweiten Website wurden mindestens 19 der 26 untersuchten Smells identifiziert. Zusätzlich wurden durchschnittlich 0,33 Instanzen von Code Smells je Zeile CSS-Code aufgefunden.

Bessghaier et al. untersuchten Code Smells im serverseitigen Code von Web-Anwendungen [BOM20]. Dafür wurden in fünf PHP-Web-Anwendungen zwölf Code Smells mithilfe des Tools *PHPMD* identifiziert. Die Smells wurden dabei sowohl hinsichtlich ihrer Verbreitung über verschiedene Klassen hinweg als auch ihrer absoluten Häufigkeit untersucht. Den Code Smells *High Method Complexity*, *High NPath Complexity* und *Long Method* wurde dabei gleichzeitig eine hohe Verbreitung als auch eine hohe Häufigkeit nachgewiesen. Zusätzlich zeigte sich, dass Klassen, welche Code Smells enthalten, deutlich häufiger in der weiteren Entwicklung bearbeitet wurden und somit einen höheren Wartungsaufwand besitzen. Wie sehr zu der Häufigkeit von Änderungen beigetragen wird, variiert der Untersuchung nach zwischen den verschiedenen Smells.

Nguyen et al. untersuchten den Ursprung von „eingebetteten Code Smells“ im Web [NNN+12]. Dafür wurde PHP-Code auf das Generieren von HTML-, CSS- und JavaScript-Smells hin analysiert. Mithilfe des eigenständig entwickelten Tools *WebScent* kann der Ursprung dieser Smells im serverseitigen Code lokalisiert werden. Die Anzahl untersuchter Smells und Anwendungen lag bei jeweils sechs. Es zeigte sich, dass die PHP-Dateien, welche Code Smells verursachen, generell eine unterdurchschnittliche Qualität aufwiesen und häufiger verändert wurden als andere Dateien.

Eine rein numerische Betrachtung der vergleichbaren Analysen (⇒ Tabelle 3.1) zeigt, dass durchschnittlich 803,5 Web-Anwendungen in den bestehenden Arbeiten untersucht wurden. Da in fünf der acht beschriebenen Arbeiten 15 oder weniger Anwendungen analysiert wurden und vor allem eine Arbeit den Durchschnitt deutlich nach oben korrigiert, liegt der Median mit 13 deutlich unter dem Mittelwert. Etwas ausgeglichener ist die Anzahl untersuchter Code Smells, die auffallend häufig bei 12 oder 13 und auch im Durchschnitt bei 13,25 liegt. Zusätzlich fällt auf, dass in den untersuchten Arbeiten immer nur Code Smells exakt einer Sprache analysiert wurden. Diese numerische Betrachtung ermöglicht die Einordnung des Umfangs *dieser* Arbeit in das Feld bestehender Arbeiten (⇒ Kapitel 7.1, 8.2, 9).

Hinsichtlich der drei nutzerseitigen Sprachen (somit die PHP-Arbeiten ausgenommen) zeigt sich eine Dominanz von Arbeiten im Themengebiet der Web-Entwicklung, welche sich mit Code Smells in JavaScript beschäftigen. Mit den Arbeiten von Bleisch [Ble18] und Gharachorlu [Gha14] konnten zwei Arbeiten, welche sich mit CSS-Smells beschäftigen, identifiziert werden – wobei die Arbeit von Bleisch allerdings auch sehr stark auf der von Gharachorlu aufbaut.

Arbeit	Web-Apps	Smells	Sprache(n)
[FM13]	11	13	JavaScript
[AC20]	361	13	JavaScript
[SMKA17]	5	12	JavaScript
[JKA19]	15	12	JavaScript
[Ble18]	5.525	12	CSS
[Gha14]	500	26	CSS
[BOM20]	5	12	PHP
[NNN+12]	6	6	PHP
$\bar{x}$	803,5	13,25	1
$\tilde{x}$	13	12	1
MIN	5	6	1
MAX	5.525	26	1

Tabelle 3.1.: Anzahl untersuchter Web-Apps und Smells in vergleichbaren Analysen

$\bar{x}$: Mittelwert, $\tilde{x}$: Median, MIN: Minimalwert, MAX: Maximalwert

Abschließend ist zu erwähnen, dass *dieser* Arbeit eine Studie mit dem Titel „Analyse der Nutzung von veraltetem HTML-Code im World Wide Web“ vorausging [Sch21]. In der Studie wurde eine vergleichbare Erhebung von Daten mithilfe von Web Crawlern und Web Scrapern realisiert. Als Untersuchungsgegenstand wurden allerdings nicht Code Smells, sondern veraltete HTML-Elemente gewählt, welche als Indikator für ausgebliebenes Refactoring von Websites gewertet wurden. Dass Analysen von Code Smells aussagekräftigere und relevantere Ergebnisse liefern würden, war ein grundlegender Aspekt der Reflexion der Studie, welcher zusätzlich zum Ausgangspunkt für diese Arbeit wurde.

3.2. Verwandte Tools

Aus den theoretischen Erkenntnissen über Code Smells resultiert die Verfügbarkeit zahlreicher Tools, um Entwickler*innen bei deren Identifikation und Behebung zu unterstützen. Die Tools nutzen dafür verschiedene Erkennungsstrategien, welche bereits im Rahmen von Kapitel 2.2 vorgestellt wurden.

Rasool und Arshad liefern einen ausführlichen Überblick und Vergleich von 21 Erkennungstools für Code Smells außerhalb des Themengebiets der Web-Entwicklung [RA15]. Besonders auffällig ist dabei der Fokus auf die Programmiersprache Java, welche von allen 21 Tools unterstützt wird. Einige der Tools unterstützen zusätzlich die Sprachen C, C++, C# oder Delphi. Ein Großteil der Tools (ca. 76%) besitzt zusätzlich externe Abhängigkeiten, da sie etwa in eine Entwicklungsumgebung eingebettet werden müssen.

Mit Blick auf die Web-Entwicklung gibt es nur wenige Tools zur Erkennung von Code

Smells, welche in wissenschaftlichem Kontext entwickelt oder genutzt wurden. *JSNose* identifiziert 13 Code Smells in JavaScript und zeichnet sich durch die Kombination von statischer und dynamischer Analyse aus [FM13]. $TAJS_{lint}$ erkennt 14 JavaScript-Code-Smells und besitzt eine vergleichsweise hohe Präzisionsrate von 98% [AL20]. Das Framework *CSSNose* zeichnet sich ebenfalls durch eine hohe Präzisionsrate aus; es identifiziert 26 Code Smells und Fehler in CSS [Gha14]. *WebScent* erkennt sechs eingebettete Code Smells im serverseitigen PHP-Code mit Fokus auf die mangelhafte Umsetzung der Separation of Concerns [NNN+12].

Zusätzlich wird in wissenschaftlichen Arbeiten von der Entwicklung von Tools berichtet, welche jedoch nicht fortlaufend nutzbar sind. So entwickelten Johannes et al. ein namenloses und nicht öffentlich verfügbares Framework zum Identifizieren von zwölf Code Smells in JavaScript [JKA19]. Saboury et al. präsentieren ein Framework, welches ebenfalls zwölf JavaScript-Smells erkennen kann [SMKA17]. Der Code wurde zum Nachvollziehen der Arbeit zwar veröffentlicht, das Framework ist aufgrund des Fehlens näherer Anleitungen jedoch nicht ohne weiteres anwendbar.

Trotz der engen Zusammenarbeit von HTML, CSS und JavaScript, identifiziert keines der im wissenschaftlichen Kontext genutzten oder entwickelten Tools Code Smells in mehr als einer Sprache.

Außerdem gibt es Tools, deren Entwicklung und Anwendung nicht wissenschaftlich begleitet wurde. Im Bereich der Open-Source-Software gibt es mit *JSLint* [Cro], *ESLint* [ESL], *Codehawk* [Cod], *Flow* [Fac] und *JSHint* [Kov] Tools zur statischen Validation und Analyse von JavaScript-Code, welche auch auf ausgewählte Code Smells hinweisen. Im Bereich kommerzieller Software ist *SonarQube* zu erwähnen [Sona]. Das Programm unterstützt die Identifizierung von Bugs und Code Smells in 29 Sprachen, inklusive HTML, CSS und JavaScript.

Insgesamt lässt sich aus den zusammengetragenen Tools hinsichtlich der Web-Entwicklung ein klarer Fokus auf die Sprache JavaScript ableiten. Im Rahmen dieses Kapitels konnten zehn Tools mit JavaScript-Unterstützung vorgestellt werden – jedoch nur zwei für CSS und nur ein einziges mit der Möglichkeit HTML-Smells zu identifizieren. Diese Beobachtung stützt die in Kapitel 1.1 aufgestellte These einer „Forschungslücke“ und der unzureichenden Gesamtbetrachtung aller drei Web-Sprachen hinsichtlich Code Smells in der Praxis.

4. Vorgehen

Nach dem Zusammentragen der notwendigen theoretischen Grundlagen, finden diese nun für die eigenständigen Arbeitsschritte Anwendung. Das folgende Vorgehen ist dabei geplant und zusätzlich kompakt in Abbildung 4.1 dargestellt.

Während der Analyse der Fachliteratur wurden 66 verschiedene Code Smells, welche sich potenziell auf das Web übertragen lassen, identifiziert. Diese werden nun genauer untersucht, um festzustellen, ob sie tatsächlich für die Web-Entwicklung relevant sind und ob sie sich mit den Mitteln der statischen Code-Analyse identifizieren lassen (⇒ Kapitel 5.1). Aus dieser Analyse heraus resultiert eine finale Teilmenge an Code Smells, deren Erkennung in dem geplanten Tool implementiert wird und welche somit auch Bestandteil der geplanten Verbreitungsanalyse werden.

Nach der Identifikation dieser Teilmenge an Code Smells werden diese nochmals genauer betrachtet. Dabei liegt der Fokus auf verschiedenen Aspekten, welche für die weiteren Arbeitsschritte relevant sind. Lassen sich die jeweiligen Smells auf die Sprachen HTML, CSS und JavaScirpt übertragen? Welche Refactorings sind für diese Code Smells empfohlen? Anhand welcher Metriken lassen sich die Smells identifizieren und was sind geeignete Schwellenwerte? Zusätzlich werden gemeinsame Eigenschaften einiger dieser Code Smells identifiziert (⇒ Kapitel 5.1) und den jeweiligen Smells in Form von Labels zugeordnet. Die Ergebnisse dieser Analysen werden kompakt in den Kapiteln 5.2 bis 5.6 präsentiert.

Im Anschluss ist die erste Phase der Entwicklung (⇒ Kapitel 6) geplant. Um die geplante Verbreitungs-Analyse durchführen zu können, wird ein Web Crawler (⇒ Kapitel 6.1) benötigt, welcher die URLs von den angestrebten $\geq$10.000 Websites zusammentragen kann. Das Analyse-Tool *WebDeo* soll die URL-Sammlung anschließend einlesen können und die zugehörigen Websites nacheinander auf das Vorhandensein der Code Smells untersuchen.

Anschließend beginnt die Entwicklungsphase von *WebDeo* mit der Planung der System-Architektur (⇒ Kapitel 6.2). Es folgt die Implementation der Erkennung für die verschiedenen Code Smells unter Einsatz der Mittel der statischen Code-Analyse (⇒ Kapitel 6.3). Um die Funktionalität der Erkennung innerhalb dieses Ein-Personen-Projektes bestmöglich sicherzustellen, ist die Entwicklung nach dem Test-First-Ansatz geplant (⇒ Kapitel 6.3.1). Zusätzlich sind nach der initialen Entwicklung der Code-Smell-Erkennungen (⇒ Kapitel 6.3.4) Testläufe auf zufälligen Websites geplant, um weitere Fehler oder Schwächen, vor dem Start der groß angelegten Verbreitungsanalyse, zu identifizieren und auszubessern. Sobald das Tool einsatzbereit ist, wird es auf die von dem Crawler zusammengetragenen Websites angewendet.

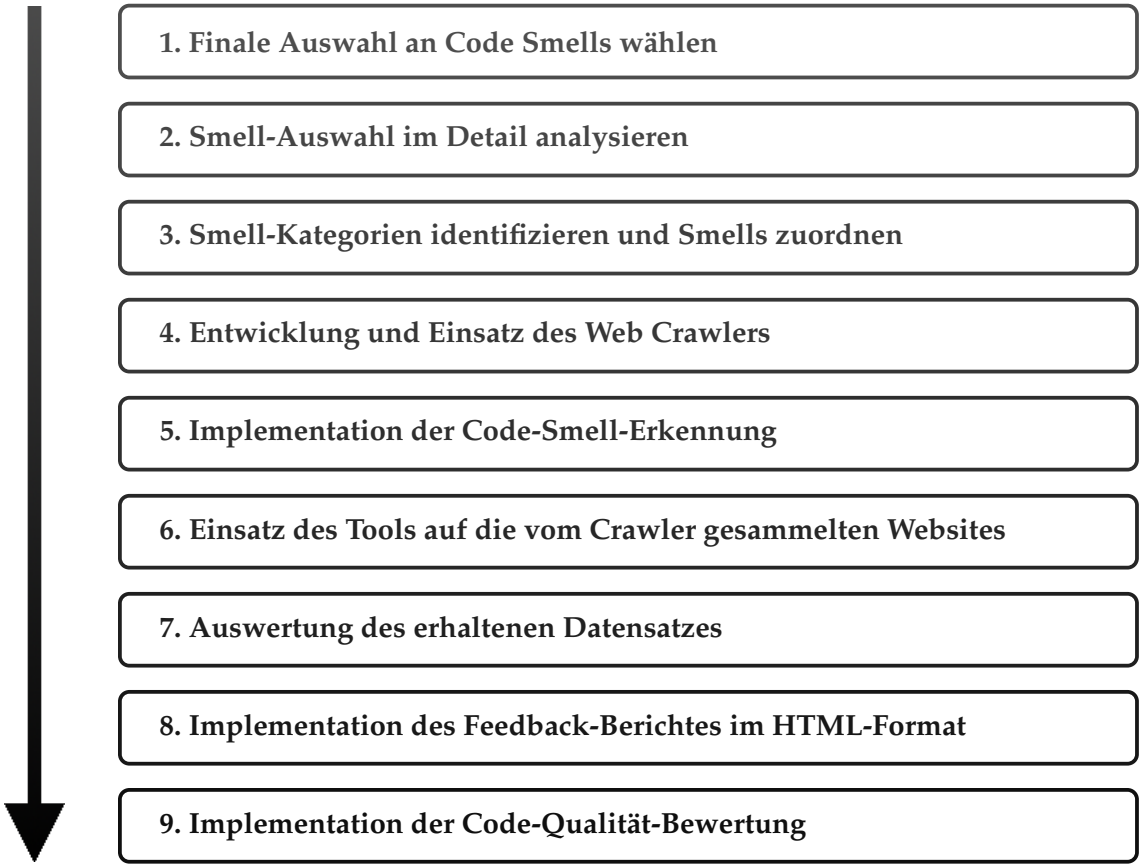


Abbildung 4.1.: Kompakte Übersicht der geplanten Arbeitsschritte

Um die Auswertung des umfangreichen Datensatzes zu erleichtern, soll ein Hilfstool entwickelt werden, welches das ursprüngliche Output-Format (JSON-Datei) so umstrukturiert, dass die Daten ohne große Umstände in Microsoft Excel importiert werden können. In Excel findet anschließend die eigentliche Auswertung der Daten statt. Dafür werden diese in sinnvolle Views unterteilt und die Kennwerte zur Beantwortung der Forschungsfragen ermittelt.

Im Anschluss beginnt die zweite Implementationsphase mit der Entwicklung eines Feedback-Berichtes (⇒ Kapitel 6.4), welcher nach der Analyse einer Website generiert werden soll. In dem Feedback-Bericht werden die identifizierten Code Smells auf der Website optisch aufbereitet präsentiert, um Entwickler*innen anschließend, unter der Empfehlung geeigneter Refactorings, beim strukturierten Entfernen der Smells zu unterstützen.

Abschließend sollen die Daten zur durchschnittlichen Verbreitung der Code Smells *WebDeo* zur Verfügung gestellt werden. Darauf aufbauend soll eine Funktion zum Abgleich einer analysierten Website mit den Durchschnittswerten implementiert werden. Dieser Vergleich soll genutzt werden, um Entwickler*innen ihr individuelles Verbesserungspotenzial, aber auch ihre persönlichen Stärken, aufzeigen zu können. Die Ergebnisse dieses Abgleichs der Code-Qualität mit den ermittelten Durchschnittswerten wird optisch aufbereitet in den Feedback-Bericht aufgenommen.

5. Code-Smells in der Web-Entwicklung

5.1. Auswahl und Kategorisierung geeigneter Code Smells

Insgesamt wurden für diese Arbeit 66 Code Smells näher betrachtet. Durch die Recherche und eigenständige Analyse der verschiedenen Smells wird RQ.I inklusive ihrer Teilfragen beantwortet.

Bei den 66 untersuchten Smells handelt es sich um bekannte, sprachübergreifend akzeptierte Code Smells aus der Programmierung (Kapitel 5.2, RQ.Ia)), HTML-spezifische Code Smells (Kapitel 5.3, RQ.Ib)), CSS-spezifische Code Smells (Kapitel 5.4, RQ.Ib)), JavaScript-spezifische Code Smells (Kapitel 5.5) sowie Code Smells, welche sich aus den Kombinationsmöglichkeiten der verschiedenen Web-Sprachen ergeben (Kapitel 5.6, RQ.Ic)).

Sämtliche sprachübergreifend bekannte Code Smells wurden eigenständig hinsichtlich ihrer Übertragbarkeit auf die drei untersuchten Sprachen analysiert. Zusätzlich wurde eine Teilmenge von 39 Code Smells, deren Erkennung für die geplante Analyse und das Tool implementiert wird, begründet ausgewählt. Diese Code Smells werden im Rahmen der folgenden Unterkapitel ausführlich vorgestellt.

Tabelle 5.1 (⇒ Seite 29) listet die 27 Code Smells, welche zwar näher betrachtet wurden, deren Erkennung jedoch nicht implementiert wird. Sie wurden ebenfalls auf ihre Übertragbarkeit auf HTML, CSS und JavaScript hin analysiert und tragen somit trotz ihrer Aussortierung zur Beantwortung von RQ.Ia) bei. In der letzten Spalte der Tabelle wird beschrieben, wieso darauf verzichtet wurde, die Identifikation des jeweiligen Smells zu implementieren. Folgende Gründe haben sich dabei ergeben.

Falsch beschrieben

Diese Praktiken sind in der Literatur entweder als Code Smell beschrieben, obwohl sie sich nicht mit der Definition aus Kapitel 2.1 decken, oder sind hinsichtlich ihrer Erkennung falsch beschrieben.

Erroneous Adjoining Selectors: Das Fehlen eines Leerzeichens zwischen CSS-Selektor-Einheiten, zum Beispiel `.class1.class2` statt `.class1 .class2`, wird in der Literatur als Fehler beschrieben, welcher den Regelblock inaktiv setze [Gha14]. Tatsächlich ist diese Syntax jedoch Teil von CSS, um Elemente zu referenzieren, welche beide Selektorangaben erfüllen [w3sb]. Eine Inaktivsetzung würde ausschließlich durch das Referenzieren von HTML-Elementen, die es nicht gibt (`spandiv` statt `span div`), eintreten. In diesem Fall handelt es sich jedoch eher um einen Bug als um einen Smell.

Undoing Styles: Das Zurücksetzen von CSS-Werten auf ursprüngliche Werte

ist eine Praxis, die auf schlechten Umgang mit der Kaskadierung hindeutet [PVZ16]. Zu Unrecht werden die Werte `0` und `none` als Indikator beschrieben und in anderen Tools genutzt [Gha14, Ble18]. Tatsächlich verfehlen diese angeblichen Indikatoren die eigentliche Bedeutung des Smells deutlich [Ble18]. In seiner eigentlichen Bedeutung ist der Smell *Nicht identifizierbar*, da nicht automatisiert entschieden werden kann, ob es sinnvoll ist, einen Wert zurückzusetzen.

Geringfügige Relevanz

Diese Code Smells sind lediglich auf JavaScript anwendbar. Gleichzeitig basieren sie auf Konzepten, welche in JavaScript wenig genutzt werden oder bereits sehr spezifisch sind. Sie besitzen daher für die Web-Entwicklung eine vergleichsweise geringe Relevanz. Daher wird auf die Implementierung der Erkennung dieser Code Smells, zugunsten anderer Smells mit besserem Aufwand-Nutzen-Verhältnis, verzichtet.

Nicht übertragbar

Dieser Code Smell ist weder auf HTML noch auf CSS oder JavaScript übertragbar.

Nicht identifizierbar

Dieser Code Smell lässt sich mit den Mitteln der statischen Code-Analyse nicht automatisiert erkennen.

Spezifikation

Dieser allgemeine Code Smell wurde für die Gegebenheiten im Web spezifiziert und daher unter anderem Namen und mit inhaltlicher Anpassung näher betrachtet.
Global Data ⇒ *Excessive Global Variables*

Tabelle 5.1 ist zu entnehmen, dass bei genauer Betrachtung lediglich drei der Code Smells weder auf JavaScript noch auf HTML oder CSS übertragbar sind. 13 weitere Code Smells wurden aussortiert, da die Mittel der statischen Code-Analyse als nicht ausreichend für ihre Identifikation eingeschätzt wurden. Neun Code Smells sind grundlegend auf die Sprachen im Web übertragbar, wurden aber aufgrund geringerer Relevanz zugunsten anderer Smells aussortiert. Bei zwei Code Smells zeigten sich in der näheren Betrachtung Ungenauigkeiten in anderen Arbeiten und Tools. Der von Fowler beschriebene Code Smell *Global Data* wurde in Form des Smells *Excessive Global Variables* auf die Gegebenheiten von JavaScript hin spezifiziert.

Durch das Aussortieren der 27 in Tabelle 5.1 gelisteten Code Smells ergab sich ein finales Set von 39 Code Smells (⇒ A.1 und folgende Unterkapitel), deren Erkennung im geplanten Tool implementiert werden soll und welche nochmals genauer analysiert wurden. Die Ergebnisse dieser Analysen werden in den folgenden Unterkapiteln in farbigen Boxen kompakt präsentiert. Dabei werden auch empfohlene Refactorings sowie die gewählte Metrik zur Erkennung der jeweiligen Code Smells genannt.

Code Smell	HTML	CSS	JS	Aussortierungsgrund
Accessing the `this` Reference in Closures [FM13]	✗	✗	✓	Geringfügige Relevanz
Alternative Classes with different Interfaces [Fow20]	✗	✗	✗	Nicht übertragbar
Closures in Loops [FM13]	✗	✗	✓	Geringfügige Relevanz
Comments [Fow20]	✓	✓	✓	Nicht identifizierbar
Data Class [Fow20]	✗	✗	(✓)	Geringfügige Relevanz
Data Clumps [Fow20]	✗	✗	✓	Nicht identifizierbar
Dead Code [Mar09, FM13]	✗	✓	✓	Nicht identifizierbar
Dependency Cycles [SS18]	✗	✗	✗	Nicht übertragbar
Divergent Change [Fow20]	✓	✓	✓	Nicht identifizierbar
Erroneous Adjoining Selectors [Gha14]	✗	✓	✗	Falsch beschrieben
Extra Bind [FM13]	✗	✗	✓	Geringfügige Relevanz
Feature Envy [Fow20]	✗	✗	✓	Nicht identifizierbar
Functional Decomposition [SS18]	✗	✗	✓	Nicht identifizierbar
Global Data [Fow20]	✗	✗	✓	Spezifikation
Insider Trading [Fow20]	✗	✗	✗	Nicht übertragbar
Large Class / God Class [Fow20]	✗	✗	(✓)	Geringfügige Relevanz
Lazy Element [Fow20]	✗	✗	(✓)	Geringfügige Relevanz
Middle Man [Fow20]	✗	✗	(✓)	Geringfügige Relevanz
Mutable Data [Fow20]	✗	✗	✓	Nicht identifizierbar
Primitive Obsession [Fow20]	✗	✗	(✓)	Nicht identifizierbar
Refused Bequest [Fow20]	✗	✗	(✓)	Nicht identifizierbar
Shotgun Surgery [Fow20]	✓	✓	✓	Nicht identifizierbar
Spaghetti Code [TPB+15, SS18]	✗	✗	✓	Nicht identifizierbar
Speculative Generality [Fow20]	✗	✗	✓	Nicht identifizierbar
Temporary Field [Fow20]	✗	✗	(✓)	Geringfügige Relevanz
Undoing Styles [Gha14, Ble18]	✗	✓	✗	Nicht identifizierbar Falsch beschrieben
Variable Name Conflict in Closures [FM13]	✗	✗	✓	Geringfügige Relevanz

Tabelle 5.1.: Liste von Code Smells, welche nach näherer Analyse aussortiert wurden

Die Wahl von Schwellenwerten basiert auf vorhandenen Angaben in der referenzierten Literatur. Von den 39 Smells treten 8 in HTML, 12 in CSS und 17 in JavaScript auf. Zusätzlich werden 8 *Separation of Concerns*-Smells näher betrachtet, die in unterschiedlichen Sprachen identifiziert werden können (⇒ Kapitel 5.6).

Mit der Beantwortung von RQ.IV) wurde das Ziel formuliert, Zusammenhänge zwischen Smells mit bestimmten Eigenschaften und ihrer Häufigkeit zu identifizieren. Um eine diesbezügliche Analyse zu ermöglichen, wurden eigenständig elf Eigenschaften identifiziert und den Code Smells in Form von Labels zugewiesen. Die Bedeutung der verschiedenen Labels wird im Folgenden erklärt. A.1 stellt die Zuordnung aller Labels zu den jeweiligen Smells übersichtlich dar.

CSS	Dieser Code Smell steht in direktem Zusammenhang mit der Sprache CSS, da er entweder in CSS auftreten kann oder einen Bruch der *Separation of Concerns* von CSS und einer weiteren Sprache darstellt.
Deklaration	Dieser Code Smell tritt innerhalb der Deklarationen von CSS-Regeln auf.
Fehlerähnlich	Diese Praktik wird in der untersuchten Literatur zwar als Code Smell gelistet, kommt der Definition eines Fehlers oder Bugs jedoch sehr nahe. Häufig handelt es sich um Unachtsamkeiten, die Fehler verursachen können, aber nicht müssen.
HTML	Dieser Code Smell steht in direktem Zusammenhang mit der Sprache HTML, da er entweder in HTML auftreten kann oder einen Bruch der *Separation of Concerns* von HTML und einer weiteren Sprache darstellt.
Inhaltlich	Bei diesem Code Smell handelt es sich um eine inhaltlich unnötig komplexe Lösung, welche sich negativ auf die Übersichtlichkeit und Verständlichkeit des Programms auswirkt.
JavaScript	Dieser Code Smell steht in direktem Zusammenhang mit der Sprache JavaScript, da er entweder in JavaScript auftreten kann oder einen Bruch der *Separation of Concerns* von JavaScript und einer weiteren Sprache darstellt.
Optisch	Dieser Code Smell trägt zur Komplexität des Programms bei und wirkt sich negativ auf dessen Verständlichkeit aus, da der Code optisch unübersichtlich formatiert oder entwickelt ist.
Selektor	Dieser Code Smell tritt innerhalb der Selektoren von CSS-Regeln auf.
SoC	Bei diesem Smell liegt ein Bruch der *Separation of Concerns* zwischen zwei der drei analysierten Sprachen des Webs vor.
Überflüssig	Bei diesem Smell wird das Programm durch funktionsfreien oder redundanten und somit überflüssigen Code aufgebläht.
Veraltet	Bei diesem Code Smell werden Konstrukte eingesetzt, die veraltet sind und nicht mehr genutzt werden sollten.

5.2. Sprachübergreifend anerkannte Code Smells aus der Programmierung

In diesem Unterkapitel werden die Ergebnisse der Analyse von 14 Code Smells präsentiert, welche hinreichend für mehrere Programmiersprachen beschrieben und darin identifiziert wurden. Es handelt sich somit ausdrücklich nicht um sprachspezifische Smells, die sich aus den Besonderheiten einzelner Sprachen ergeben. Diesen Smells kommt daher eine breite Anerkennung und vergleichsweise hohe Bekanntheit zu.

Aufgrund dieser Gegebenheit ist es wenig überraschend, dass sich alle 14 Smells auch auf die untersuchte Programmiersprache JavaScript übertragen lassen. Lediglich bei der Erkennung des Smells *Mysterious Name* gibt es Limitationen, welche jedoch den Möglichkeiten der statischen Code-Analyse geschuldet sind.

Da viele Konzepte aus Programmiersprachen nicht in HTML oder CSS enthalten sind, lassen sich nur wenige (jeweils drei) dieser Code Smells auf diese Sprachen übertragen.

Gemeinsam mit den aussortierten Smells außerhalb der Kategorie *Nicht übertragbar* sowie den aussortierten CSS-Smells *Erroneous Adjoining Selectors* und *Undoing Styles* (⇒ Tabelle 5.1) stellt diese Auswahl an Code Smells das Ergebnis zu RQ.Ia) dar.

Ergebnis RQ.Ia)

Welche klassischen Code Smells lassen sich auf die webspezifischen Sprachen HTML, CSS und JavaScript übertragen?

Von den 25 aussortierten Code Smells aus klassischen Programmiersprachen lassen sich 3 auf HTML, 6 auf CSS und 22 auf JavaScript übertragen. Um welche Smells es sich handelt, ist Tabelle 5.1 zu entnehmen.
Von den 14 detaillierter analysierten Smells sind jeweils 3 auf HTML und CSS sowie alle 14 auf JavaScript übertragbar. Um welche Smells es sich handelt, zeigen die folgenden Analyse-Ergebnisse.

Assignment in Conditional Statement

Beschreibung	Benutzung eines Zuweisungsoperators wie `=` anstelle eines Vergleichsoperators wie `==` in einer Bedingung, zum Beispiel `if (a = b)`. [JKA19]		
Refactorings	Zuweisungsoperator durch Vergleichsoperator ersetzen		
Anwendbarkeit	HTML	✗	Keine Bedingungen
	CSS	✗	Keine Bedingungen
	JavaScript (JS)	✓	Durch Fehlen eines Compilers ermöglicht
Typ	`boolean`		
Labels	Fehlerähnlich JavaScript		

Chained Methods

Beschreibung	Übermäßige Aneinanderkettung von Methodenaufrufen und ihrer Rückgabewerte durch Punktnotation. [AL20, JKA19]
Refactorings	Hide Delegate, Extract Method [Shv19]
Anwendbarkeit	HTML ✗ Keine Methoden CSS ✗ Keine Methoden JS ✓
Metrik	Number of Chained Methods (NCM)
Schwellenwert	$NCM > 5$ [AL20]
Labels	Inhaltlich JavaScript Optisch

Commented-Out Code

Beschreibung	Vorhandensein von Code-Strukturen, die zur Aufbewahrung auskommentiert wurden. Durch Repositories ist das Speichern von Code auf diese Art unnötig. [Mar09]
Refactorings	Auskommentierten Code entfernen
Anwendbarkeit	HTML ✓ CSS ✓ JS ✓
Typ	`boolean`
Labels	CSS HTML JavaScript Optisch Überflüssig

Complex Switch Case

Beschreibung	Übermäßige Anzahl an Case-Statements in einem Switch-Statement. [JKA19]
Refactorings	Extract Method, Replace Type Code with Subclasses [Shv19]
Anwendbarkeit	HTML ✗ Keine Switch-Case-Statements CSS ✗ Keine Switch-Case-Statements JS ✓
Metrik	Number of Case Statements (NCS)
Schwellenwert	$NCS > 6$ [JKA19]
Labels	JavaScript Optisch

Depth

Beschreibung	Hohe Anzahl von ineinander geschachtelten Blöcken wie Fallunterscheidungen und Schleifen. [JKA19]
Refactorings	Replace Nested Conditional with Guard Clauses, Extract Method [Shv19]
Anwendbarkeit	HTML ✗ Einrückung und tiefe Verschachtelung von HTML-Tags unvermeidbar CSS ✗ Regel-Syntax schließt tiefe Schachtelung aus. JS ✓
Metrik	Number of Nested Blocks (NNB)
Schwellenwert	$NNB > 3$ [Sonb]
Labels	JavaScript Optisch

Duplicated Code

Beschreibung	Vorhandensein von zwei oder mehr identischen Code-Fragmenten. [Fow20]
Refactorings	Extract Method, Extract Superclass, Extract Class [Shv19]
Anwendbarkeit	HTML ✗ Code-Wiederholung ist üblich, wenn Inhalte mehrfach auf einer Website erscheinen CSS ✓ Anwendbar auf identische Selektoren oder Deklarationsabschnitte JS ✓ Mehrfach auftretende Blöcke von mindestens 3 aufeinanderfolgender, identischer Zeilen
Typ	`boolean`
Labels	CSS JavaScript Optisch Überflüssig

Empty Catch

Beschreibung	Vorhandensein eines leeren Catch-Blockes. Indiz für schlechten Umgang mit Exceptions. [HLMHC17]
Refactorings	try-Block entfernen oder auf Fehler in Callbacks prüfen [HLMHC17]
Anwendbarkeit	HTML ✗ Keine Exceptions CSS ✗ Keine Exceptions JS ✓
Metrik	Lines of Code im Catch-Block (LOC_CATCH)
Schwellenwert	$LOC\_CATCH < 1$ [FM13]
Labels	JavaScript Überflüssig

Lengthy Lines

Beschreibung	Zu lange Zeilen schaden der Übersichtlichkeit und Verständlichkeit von Code. [JKA19]
Refactorings	Zeilen aufteilen
Anwendbarkeit	HTML ✓ CSS (✓) Längenunabhängig: Eine Deklaration je Zeile JS ✓
Metrik	Number of Characters je Zeile (NOC_LINE) (HTML, JS) Number of Declarations je Zeile (NOD_LINE) (CSS)
Schwellenwert	$NOC\_LINE > 80$ (HTML, JS) [JKA19] $NOD\_LINE > 1$ (CSS)
Labels	CSS Deklaration HTML JavaScript Optisch

Long Method

Beschreibung	Zu umfangreiche Methoden schaden der Übersichtlichkeit und sind Indiz dafür, dass die Methode mehr als eine konkrete Aufgabe erfüllt. [FM13, SMK17, RA15]
Refactorings	Extract Method [Shv19]
Anwendbarkeit	HTML ✗ Keine Methoden CSS ✗ Keine Methoden JS ✓
Metrik	Lines of Code je Methode (LOC_METHOD)
Schwellenwert	$LOC\_METHOD > 30$ (Mittelwert von [FM13] und [Shv19])
Labels	JavaScript Optisch

Long Parameter List

Beschreibung	Zu viele Parameter machen Methoden unübersichtlich und weisen auf mehr als eine Aufgabe hin. [Fow20, Shv19]
Refactorings	Introduce Parameter Object [Fow20]
Anwendbarkeit	HTML ✗ Keine Parameter CSS ✗ Keine Parameter JS ✓
Metrik	Number of Parameters (NOP)
Schwellenwert	$NOP > 4$ (Mittelwert von [FM13] und [Shv19])
Labels	Inhaltlich JavaScript

Mysterious Name

Beschreibung	Es sollten aussagekräftige Bezeichner gewählt werden, um Code deutlich verständlicher zu machen. [Fow20]
Refactorings	Change Function Decleration, Rename [Fow20]
Anwendbarkeit	HTML (✓) ID- und Klassennamen CSS ✗ Keine Namenswahl JS (✓) Klassen- und Methodennamen, Variablen (⇒ Limitationen)
Limitationen	⇒ A.2, S. XVII
Metrik	Number of Characters je Name (NOC_NAME)
Schwellenwert	$NOC\_NAME < 2$
Labels	HTML JavaScript

Nested Callbacks

Beschreibung	Ineinander geschachtelte Callback-Funktionen schaden der Verständlichkeit des Codes. [JKA19, FM13]
Refactorings	Extract Method [FM13]
Anwendbarkeit	HTML ✗ Keine Callbacks CSS ✗ Keine Callbacks JS ✓
Beispiel	⇒ A.2, S. XVIII
Metrik	Callback Depth (CBD)
Schwellenwert	$CBD > 3$ [FM13]
Labels	Inhaltlich JavaScript Optisch

Repeated Switches

Beschreibung	Gleiche Bedingungen in mehreren Switch-Statements. [Fow20]
Refactorings	Extract Method, Replace Conditional with Polymorphism [Shv19]
Anwendbarkeit	HTML ✗ Keine Switch-Case-Statements CSS ✗ Keine Switch-Case-Statements JS ✓
Typ	`boolean`
Labels	JavaScript Optisch Überflüssig

This Assign

Beschreibung	Speichern einer `this`-Variable in einer anderen Variable zum Umgehen von Sichtbarkeits-Einschränkungen. [JKA19]
Refactorings	`.bind(this)` oder Pfeilfunktionen nutzen (in JavaScript) [JKA19]
Anwendbarkeit	HTML ✗ Keine Variablen CSS ✗ Keine Variablen JS ✓
Beispiel	⇒ A.2, S. XVIII
Typ	`boolean`
Labels	JavaScript

5.3. HTML-spezifische Code Smells

Diese Code Smells treten explizit in der Strukturierungssprache HTML auf, da sie mit sprachspezifischen Komponenten im Zusammenhang stehen. Daher lassen sich diese Smells auch nicht auf andere Sprachen übertragen. Insgesamt wurden fünf dieser HTML-spezifischen Code Smells analysiert. Gemeinsam mit dem Ergebnis aus Kapitel 5.4 stellt diese Smell-Auswahl das Ergebnis zu RQ.Ib) dar.

Teilergebnis RQ.Ib)

Welche weiteren Code Smells existieren spezifisch für die Strukturierungssprache HTML und die Designsprache CSS?

Es wurden 5 HTML-spezifische Code Smells identifiziert. Die folgenden Analyse-Ergebnisse zeigen auf, um welche Smells es sich dabei handelt.

Deprecated Attributes

Beschreibung	Zahlreiche Attribute (⇒ A.2, S. XVI) wurden generell oder für ausgewählte Tags als „deprecated“ (veraltet) eingestuft. Ihre fortlaufende Nutzung kann zu einer Vielzahl unterschiedlicher Nachteile führen. [Sch21]
Refactorings	Attribut entfernen, Funktionalität über CSS / JavaScript / alternative HTML-Lösung sicherstellen
Typ	`boolean`
Labels	HTML Veraltet

Deprecated Tags

Beschreibung	Einige Tags (⇒ A.2, S. XVI) wurden durch andere Tags ersetzt oder als „deprecated“ eingestuft, da ihre Funktion durch die Kombination anderer Komponenten umgesetzt werden sollte. [Sch21]
Refactorings	Tag durch moderne Lösung ersetzen
Typ	`boolean`
Labels	HTML Veraltet

Opening Tag missing Closing Tag

Beschreibung	Einige HTML-Elemente bestehen aus einem öffnenden und einem schließenden Tag. Fehlende schließende Tags führen zu Darstellungsfehlern und unübersichtlichem Code. [NNN+12]
Refactorings	Schließendes Tag hinzufügen
Typ	`boolean`
Labels	Fehlerähnlich HTML Optisch

Skipped Headings

Beschreibung	Überschriften sollten zur korrekten Strukturierung und Interpretation durch Suchmaschinen stets in korrekter Ordnung sein. Je Seite sollte es exakt ein `h1`-Element geben und anschließend keine Lücken, etwa ein `h3`-Element ohne vorheriges `h2`-Element. [w3se]
Refactorings	Überschriften-Ordnung anpassen
Typ	`boolean`
Labels	HTML

Uppercase Tags

Beschreibung	Zwecks Einheitlichkeit sollten HTML-Tags stets entweder groß oder klein geschrieben werden. Die Kleinschreibung hat sich als gelebter Standard etabliert (`<p>` statt `<P>`). [Har12]
Refactorings	Großgeschriebene Tags zu kleingeschriebenen ändern
Typ	`boolean`
Labels	HTML

5.4. CSS-spezifische Code Smells

Die folgenden neun Code Smells beziehen sich auf sprachspezifische Komponenten von CSS. Dementsprechend können diese Smells auch nicht in anderen Sprachen auftreten und identifiziert werden. Gemeinsam mit dem Ergebnis aus Kapitel 5.3 sowie den aussortierten CSS-Smells *Erroneous Adjoining Selectors* und *Undoing Styles* stellt diese Smell-Auswahl das Ergebnis zu RQ.Ib) dar.

Teilergebnis RQ.Ib)

Welche weiteren Code Smells existieren spezifisch für die Strukturierungssprache HTML und die Designsprache CSS?

Es wurden 9 CSS-spezifische Code Smells detailliert analysiert. Die folgenden Boxen zeigen auf, um welche Smells es sich dabei handelt.
Zusätzlich wurden mit *Erroneous Adjoining Selectors* und *Undoing Styles* zwei weitere CSS-spezifische Smells identifiziert, welche aufgrund fehlerhafter Beschreibungen in der Literatur nicht näher betrachtet wurden.

Empty Rules

Beschreibung	Die Eröffnung einer Regel über einen Selektor, welche allerdings keine Deklarationen enthält, bläht den Code unnötig auf. [Ble18]
Refactorings	Regel entfernen
Typ	`boolean`
Labels	CSS Optisch Überflüssig

High Specificity Values

Beschreibung	Selektoren, die zu spezifisch sind, reagieren unflexibel auf Änderungen des HTMLs. [Ble18, Gha14]
Refactorings	Zielelemente über eigene Klassen oder IDs referenzieren
Metriken	Number of ID Selectors (NIS) Number of Class Selectors (NCLS) Number of Element Selectors (NES)
Schwellenwerte	$NIS > 1 \vee NCLS > 2 \vee NES > 3$ [Gha14]
Beispiele	⇒ A.2, S. XVI
Labels	CSS Selektor

Properties with Hard-Coded Values

Beschreibung	Durch die Vielzahl verschiedener Bildschirmgrößen sollten stets relative (`%`, `em`, ...) statt absolute (`px`, `cm`, ...) CSS-Einheiten genutzt werden. [Gha14, Ble18]
Refactorings	Absolute Werte in relative Werte umwandeln
Typ	`boolean`
Labels	CSS Deklaration

Selectors with ID and Class or Element

Beschreibung	Selektoren, die mit einer ID abschließen, referenzieren bereits ein eindeutiges HTML-Objekt. Vor der ID sollte keine Klasse oder kein Element mehr stehen. [Ble18]
Refactorings	Klassen- oder Elementreferenz entfernen
Beispiel	⇒ A.2, S. XVIII
Typ	`boolean`
Labels	CSS Inhaltlich Selektor Überflüssig

Too general Selectors

Beschreibung	Die Benutzung ausgewählter alleinstehender Selektoren (⇒ A.2, S. XVIII) sollte vermieden werden, da sie zu unspezifisch sind und die schlechte Praxis *Undoing Styles* [PVZ16] provozieren. [Gha14]
Refactorings	Spezifischere Elemente ansprechen
Typ	`boolean`
Labels	CSS Selektor

Too long Rules

Beschreibung	Regeln mit zu hoher Anzahl an Deklarationen schaden der Übersicht und Verständlichkeit. [Ble18, Gha14]
Refactorings	Regel mit zueinander passenden Deklarationen extrahieren
Metrik	Number of Values (NOV) je Regel
Schwellenwert	$NOV > 5$ [Ble18, Gha14]
Labels	CSS Optisch

Too much Cascading

Beschreibung	Bestehen Selektoren aus zu vielen Einheiten, gelten sie als zu spezifisch und sind unflexibel bezüglich Änderungen des HTMLs. [Ble18, Gha14]
Refactorings	Zielelemente über eigene Klassen oder ID referenzieren
Metrik	Number of Selector Units (NOSU)
Schwellenwert	$NOSU > 4$ [Ble18, Gha14]
Labels	CSS Selektor

Universal Selectors

Beschreibung	Die Nutzung des Universal-Selektors `*` sollte vermieden werden, da er zu unspezifisch ist und das Rendern der Websites verlangsamt. [Ble18]
Refactorings	Spezifischere Selektoren wählen
Typ	`boolean`
Labels	CSS Selektor

Usage of !important

Beschreibung	Mit `!important` wird die Priorität von CSS-Regeln übergangen. Eine eigentlich übersichtliche und verständliche Ordnung wird dadurch komplex und führt zu unerwünschten Nebeneffekten. [PVZ16, w3sc]
Refactorings	Gewünschte Elemente über spezifischere Selektoren ansprechen
Typ	`boolean`
Labels	CSS Deklaration

5.5. JavaScript-spezifische Code Smells

Zusätzlich zu den HTML- und CSS-spezifischen wurden auch drei JavaScript-spezifische Code Smells identifiziert. Sie resultieren aus den Besonderheiten, die JavaScript im Vergleich zu anderen Programmiersprachen besitzt.

Wenngleich keine der gestellten Forschungsfragen JavaScript-spezifische Code Smells thematisiert, sollte deren Vorhandensein dennoch anerkannt werden. Entwickler von Tools zur Identifikation von JavaScript-Smells sollten sich somit nicht nur auf Smells, die sprachübergreifend bekannt sind (⇒ Kapitel 5.2), fokussieren.

Argument Count Mismatch

Beschreibung	Aufgrund des fehlenden Kompiliervorgangs können Methoden mit einer von ihrer Deklaration abweichenden Anzahl an Parametern aufgerufen werden. Dies kann zu Fehlern und unvorhergesehenen Resultaten führen. [AL20]
Refactorings	Methodenaufruf korrigieren
Typ	`boolean`
Labels	Fehlerähnlich JavaScript

Array Length Assignment

Beschreibung	In JavaScript kann der Wert des `length`-Attributes von Arrays gesetzt werden. Dies führt häufig zu unerwünschten und kaum nachvollziehbaren Veränderungen an dem Zustand des Arrays. [AL20]
Refactorings	Änderungen an Kopie des Arrays vornehmen
Limitationen	Über die statische Code-Analyse können nur Array-Variablen erkannt werden, denen explizit ein Array zugewiesen wurde (`x = new Array()`, `x = [1, 2]`). Sollte etwa der Rückgabewert einer Methode ein Array sein und dieses zugewiesen werden, kann dies nicht erkannt werden (`x = str.split()`).
Typ	`boolean`
Labels	Fehlerähnlich JavaScript

Excessive Global Variables

Beschreibung	Durch die seltene Nutzung von Klassen werden in JavaScript häufig globale Variablen genutzt. Gleichzeitig sind diese sogar dateiübergreifend zugreifbar und führen häufig zu Namenskonflikten. Daher sollte die Anzahl an globalen Variablen so niedrig wie möglich gehalten werden. [FM13, SMK17]
Refactorings	Globale Variablen als Attribute in globalem Objekt zusammenfassen [FM13]
Metrik	Number of Global Variables (NGV)
Schwellenwert	$NGV > 10$ [SMK17]
Labels	Fehlerähnlich JavaScript

5.6. Code Smells durch Kombination der Sprachen im Web

HTML, CSS und JavaScript arbeiten auf Websites eng zusammen. Daher gibt es auch viele Möglichkeiten, die Sprachen auf Code-Ebene miteinander zu kombinieren. Mit Blick auf das grundlegende Prinzip der *Separation of Concerns* sollte jedoch die bestmögliche Trennung angestrebt werden: Ausschließlich im HTML-Dokument sollten eigenständige CSS- und JavaScript-Dateien referenziert werden. Eine Durchmischung der Syntax sollte stets verhindert werden.

Bei den verschiedenen Möglichkeiten, gegen dieses Prinzip zu verstoßen, handelt es sich somit um web-spezifische Code Smells, die sich aus der Kombination der drei Sprachen ergeben. Diese Code Smells liefern eine Antwort auf RQ.Ic). Im Rahmen dieser Arbeit wurden acht verschiedene Code Smells dieser Kategorie identifiziert, welche im Folgenden ausführlicher vorgestellt werden.

Es fällt auf, dass sich alle acht identifizierten Code Smells auf jeweils zwei der drei Sprachen beziehen. Somit wurde kein Code Smell identifiziert, bei dem HTML, CSS und JavaScript alle gleichzeitig miteinander durchmischt werden. Außerdem zeigt sich, dass es besonders viele Wege (sieben der acht identifizierten Smells) gibt, HTML auf eine unerwünschte Weise mit einer der beiden anderen Sprachen zu kombinieren.

Ergebnis RQ.Ic)

Gibt es weitere Code Smells im Web, die auf der Kombination der drei Sprachen basieren?

Ja. Obwohl HTML, CSS und JavaScript eng miteinander arbeiten müssen, sollten sie auf Code-Ebene bestmöglich separiert werden. Durch das Einhalten des Prinzips der *Separation of Concerns* wird die Wartung des Codes erleichtert und die Übersichtlichkeit des Codes bewahrt. Die unterschiedlichen Wege, gegen dieses Prinzip zu verstoßen, stellen web-spezifische Code Smells dar, die sich durch die Kombination der drei Sprachen ergeben. Im Rahmen dieser Arbeit wurden acht dieser Code Smells identifiziert, welche im Folgenden näher betrachtet werden.

CSS in HTML Style Attribute

Beschreibung	Über das `style`-Attribut wird einem spezifischem HTML-Element CSS-Code innerhalb des HTML-Codes zugewiesen. [NNN+12]
Refactorings	CSS-Code in externe CSS-Datei verschieben, Element über Selektor ansprechen
Typ	`boolean`
Identifikation	In HTML
Labels	CSS HTML SoC

CSS in HTML Style Tag

Beschreibung	CSS-Code befindet sich durch die Benutzung des `style`-Tags direkt in HTML-Dateien. [Ble18]
Refactorings	CSS-Code in externe CSS-Datei verschieben
Typ	`boolean`
Identifikation	In HTML
Labels	CSS HTML SoC

CSS in JavaScript

Beschreibung	CSS-Werte sollten nicht direkt, sondern über CSS-Klassen in JavaScript gesetzt werden. Indikator ist die direkte Manipulation des `style`-Attributs: `element.style.display = none`. [FM13, AC20]
Refactorings	CSS-Klassen einführen und über JavaScript setzen
Typ	`boolean`
Identifikation	In JavaScript
Labels	CSS JavaScript SoC

Deprecated Styling Tags

Beschreibung	Einige HTML-Tags (⇒ A.2, S. XVI) wurden als „deprecated“ eingestuft, da sie CSS-Aufgaben erfüllten. [Sch21]
Refactorings	Tag ändern und Design-Funktion über CSS definieren
Typ	`boolean`
Identifikation	In HTML
Labels	CSS HTML SoC Veraltet

HTML in JavaScript

Beschreibung	HTML-Strukturen sollten weder über die Methode `createElement()` noch durch HTML-Code-Strings in JavaScript zusammengesetzt werden. [FM13, AC20]
Refactorings	HTML-Code in Templates auslagern
Typ	`boolean`
Identifikation	In JavaScript
Beispiele	⇒ A.2, S. XVII
Labels	HTML JavaScript Optisch SoC

JavaScript in HTML Events

Beschreibung	Durch die Nutzung von Event-Attributen (⇒ A.2, S. XVII) wird JavaScript-Code direkt in HTML-Tags eingebunden oder über Methoden aufgerufen. Über Event-Listener wird eine deutlich bessere *SoC* erreicht. [FM13, NNN+12, Mar19]
Refactorings	Event-Listener in externer JavaScript-Datei nutzen
Typ	`boolean`
Identifikation	In HTML
Labels	HTML JavaScript SoC

JavaScript in HTML Links

Beschreibung	JavaScript-Code wird dem `href`-Attribut des `a`-Tags zugewiesen. [NNN+12]
Refactorings	Event-Listener in externer JavaScript-Datei nutzen
Typ	`boolean`
Identifikation	In HTML
Beispiel	⇒ A.2, S. XVII
Labels	HTML JavaScript SoC

JavaScript in HTML Script Tag

Beschreibung	JavaScript-Code befindet sich innerhalb eines `script`-Tags direkt in der HTML-Datei. [FM13, NNN+12]
Refactorings	Code in externe JavaScript-Datei verschieben und mit `src`-Attribut einbinden [NNN+12]
Typ	`boolean`
Identifikation	In HTML
Labels	HTML JavaScript SoC

6. Entwicklung

6.1. Externer Web Crawler

Als erster Schritt der Entwicklungsphase stand das Programmieren eines Web Crawlers an, welcher eigenständig und außerhalb von *WebDeo* existieren soll. Wie auch für alle anderen in dieser Arbeit gewählten Code-Komponenten fiel die Wahl der Programmiersprache dabei auf Python. Neben Gründen der persönlichen Erfahrung und Fortbildung sprach das Vorhandensein nützlicher Module im Bereich des Zugriffs und Auslesens von Websites und zur statischen Code-Analyse für diese Option. Um den Crawler optimal an die Bedürfnisse für die geplante Analyse anpassen zu können, wurde sich bewusst gegen ein existierendes Tool und für eine eigenständige Entwicklung entschieden.

Da der Web Crawler die Websites für die geplante Analyse zusammentragen sollte, waren bereits bei dessen Entwicklung wichtige Entscheidungen zu treffen. Bei diesen Entscheidungen ging es um direkte Auswirkungen auf die Qualität des Datensatzes. Die gewählten Lösungen sollten einen möglichst fehlerfreien Ablauf sicherstellen.

Um eine hohe Vielfältigkeit im Datensatz sicherzustellen, sollte der Crawler nie mehr als eine Web Page von der gleichen Domain zusammentragen (E1). Somit wird der verfälschende Einfluss von Duplikaten auf den Datensatz minimiert. Würde der Crawler etwa eine Web Page von YouTube aufrufen, würde er dutzende Verlinkungen auf weitere YouTube-Videos extrahieren. Werden diese Video-Seiten wiederum aufgerufen, wächst die Anzahl von gesammelten YouTube-URLs exponentiell. Da diese alle nach dem gleichen HTML-Template aufgebaut sind und dieselben CSS- und JavaScript-Dateien einbinden, würde *WebDeo* im geplanten Experiment fortlaufend identische Code-Abschnitte analysieren. Sie hätten somit einen hohen Anteil am Gesamtergebnis. Mit dem strengen Domain-Limit von je einer einzigen Web Page wurde sich somit für einen verlangsamten Crawling-Prozess zugunsten der Vorbeugung dieses Effektes und zur Sicherstellung der Qualität des Datensatzes entschieden.

Herausforderung H1 ⇒ Entscheidung E1
Durch einheitliche HTML-Templates und die Mehrfachverwendung von CSS- und JavaScript-Dateien würde das Analysieren von mehreren Web Pages der gleichen Domain, wie Crawler sie häufig auffinden, das Ergebnis ungewollt verfälschen.
Je Domain sollte der Crawler nur eine einzige Web Page analysieren.

In Folge der Entscheidung E1 kann im Bezug auf die durchgeführte Analyse sowohl von Ergebnissen *je Website* als auch *je Web Page* gesprochen werden. Da jede Website in

der Analyse von nur einer Web Page repräsentiert wird, entsprechen sich die Aussagen in diesem Kontext. Im weiteren Verlauf dieser Arbeit wird von Ergebnissen für *Websites* die Rede sein. Grund hierfür ist, dass große Teile von JavaScript- und CSS-Code über alle Web Pages einer Website hinweg eingebunden werden. Tatsächlich wird somit größtenteils der Code einer kompletten Website analysiert – wenngleich der Code von einer einzigen Web Page extrahiert wird. Lediglich in HTML wird tatsächlich nur der Code einer Web Page betrachtet. Aufgrund der beschriebenen Template-Problematik handelt es sich hierbei auch um die einzig sinnvolle Lösung.

Der analysierte Code-Umfang in HTML repräsentiert somit einen kleineren Ausschnitt einer Website als dies in CSS und JavaScript der Fall ist. Daher werden die Ergebnisse dieser Arbeit nicht nur *je Website* betrachtet, sondern alle analysierten Verbreitungen auch auf den Code-Umfang *je Datei*, *je Zeile* und *je Zeichen* aufgeschlüsselt.

Auf Basis der Reflexion einer vorherigen Arbeit soll weiteren Verfälschungen vorgebeugt werden, indem die Auswahl an Start-URLs für den Web Crawler zufällig gefällt wird (E2). Im damaligen Experiment zeigte sich, dass eine Auswahl von besonders populären Websites zu einem Datensatz mit überdurchschnittlicher Code-Qualität führen kann, da Websites dieser Größe von hochprofessionellen Unternehmen entwickelt werden [Sch21]. Für die Auswahl der Start-URLs wurde daher die *Random Website Machine* von *whatsmyip.org* genutzt [wha].

Herausforderung H2 ⇒ Entscheidung E2
Die direkte Auswahl von Start-URLs für den Crawler könnte die analysierte Code-Qualität ungewünscht vom Durchschnitt abweichen lassen.
Die Start-URLs werden zufällig bestimmt.

Da eine hohe Stichprobengröße von mindestens 10.000 URLs angestrebt wird, ist die Laufzeit des Crawlers aufgrund der vielen Web-Zugriffe zu berücksichtigen. Sowohl aus diesen zeitlichen Gründen als auch zur Absicherung im Fall eines unerwarteten Fehlers sollte der Fortschritt des Crawlers regelmäßig gespeichert werden. Dies ermöglicht die Fortführung des Prozesses zu einem späteren Zeitpunkt (E3).

Herausforderung H3 ⇒ Entscheidung E3
Zum Zusammentragen des großen Datensatzes wird der Crawler über längere Zeit im Einsatz sein.
Ein Mechanismus zum Speichern des Fortschrittes, welcher die Unterbrechung und Fortsetzung des Crawling-Prozess ermöglicht, wird implementiert.

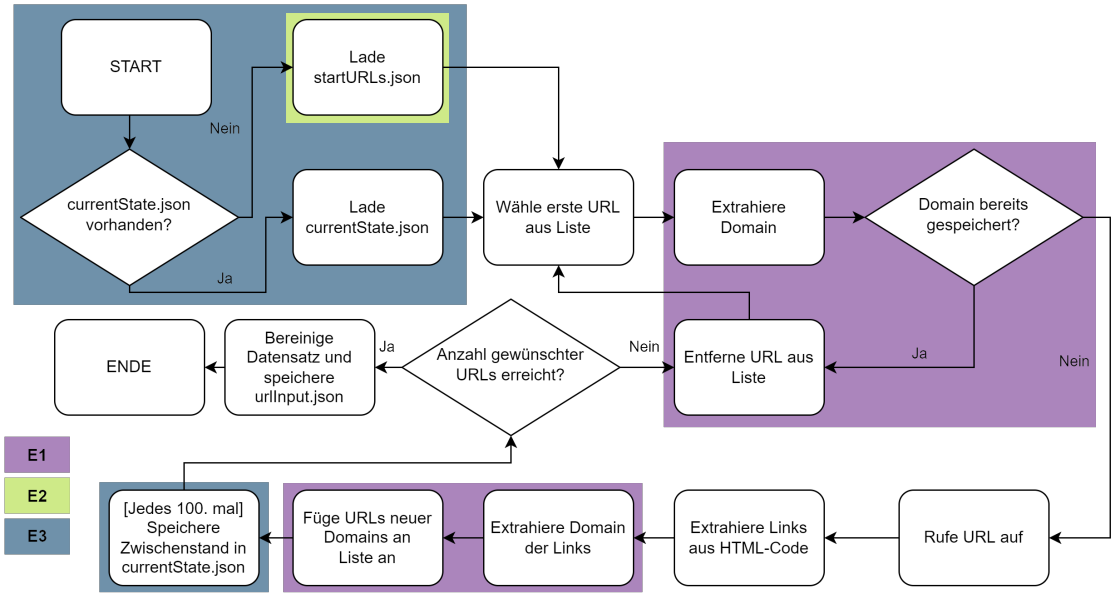


Abbildung 6.1.: Funktionsweise des entwickelten Web Crawlers unter Berücksichtigung der Entscheidungen E1, E2 und E3

Die Funktionsweise des umgesetzten Web Crawlers, unter Berücksichtigung der Entscheidungen E1, E2 und E3, ist in Abbildung 6.1 dargestellt. Als Datenformat für den Input und Output des Crawlers sowie von *WebDeo* wurde sich für JSON-Dateien entschieden. Der Crawler liest zu Beginn seiner Ausführung die nach E2 gewählten Start-URLs (`startURLs.json`) ein. Dadurch hat der Crawler eine Liste an abzuarbeitenden URLs zur Verfügung, an die er fortlaufend weitere gefundene URLs anhängt, wenn diese nicht aufgrund des Domain-Limits unmittelbar aussortiert wurden (E1). Nach jeder 100. gefundenen URL speichert der Crawler seinen Zwischenstand ab (`currentState.json`). Mithilfe dieser Datei ist es möglich, den Crawling-Prozess jederzeit zu unterbrechen und zu einem späterem Zeitpunkt weiterzuführen (E3). Dafür werden auch gefundene, aber ungeprüfte URLs sowie bereits zusammengetragene Domains zur Sicherstellung von E1 in der Datei gespeichert. Sobald die gewünschte Anzahl an URLs zusammengetragen wurde, wird der Datensatz des Crawlers um später nicht mehr benötigte Daten, zum Beispiel die gesammelten Domains, bereinigt. Es entsteht eine Inputdatei für *WebDeo* (`urlInput.json`).

6.2. WebDeo: System-Architektur

Bei der Planung der System-Architektur für das Code-Smell-Erkennungstool *WebDeo* wurden jeweils vier grundlegende funktionale Komponenten und benötigte Datenspeicher identifiziert. Die Aufgaben und die Interaktion dieser Komponenten werden im Folgenden näher erklärt. Zusätzlich ist das Zusammenspiel der Komponenten in Abbildung 6.2 in Form von einem UML-Klassendiagramm dargestellt, welches um Dateien erweitert wurde. Bei dem Diagramm handelt es sich um eine an vielen Stellen verkürzte Darstel-

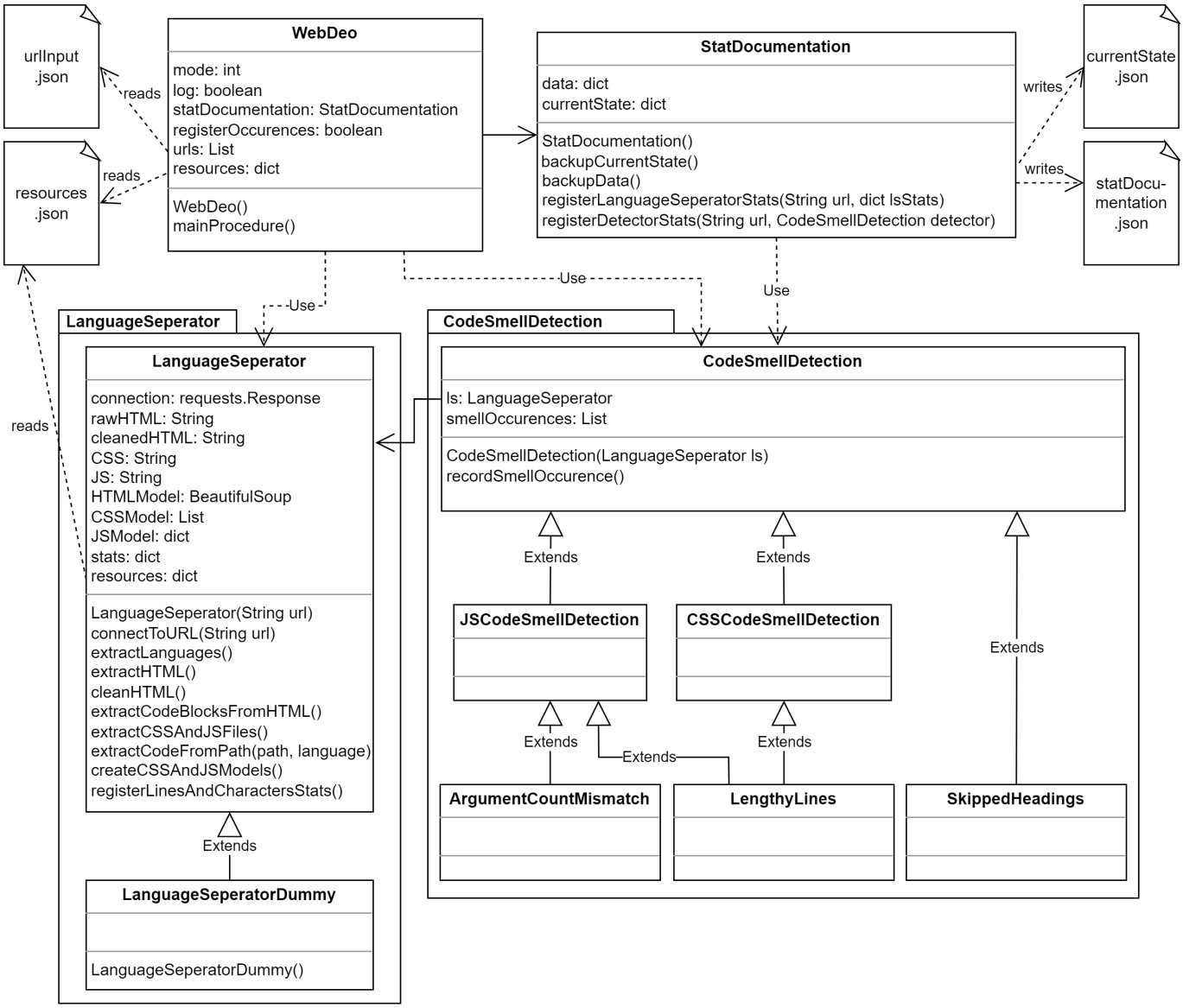


Abbildung 6.2.: Grundlegende System-Architektur von *WebDeo* (zugunsten der Übersichtlichkeit verkürzt)

lung der tatsächlichen Umsetzung des Programms. Der Fokus des gezeigten Ausschnitts liegt somit auf der übersichtlichen Darstellung der System-Architektur und erhebt nicht den Anspruch, eine annähernd vollständige Dokumentation des Programms zu sein.

Mit der Hauptklasse `WebDeo` wird der grundlegende Ablauf des Programms gestartet und konfiguriert. Über Benutzer-Eingaben kann gesteuert werden, in welchem Modus das Programm gestartet wird, also ob zum Beispiel eine umfangreiche Analyse oder die Erstellung eines Feedback-Reports vorgesehen ist. Die Klasse liest zwei JSON-Dateien ein. `urlInput.json` ist die vom Crawler zur Verfügung gestellte Sammlung von URLs für die Durchführung einer umfangreichen Verbreitungsanalyse von Code Smells im Web. In `resources.json` werden verschiedene Informationen gespeichert, die über das gesamte Programm hinweg benötigt werden, zum Beispiel welche Code Smells in

welchen Sprachen analysiert werden sollen. In der Methode `mainProcedure()` iteriert `WebDeo` über die eingelesenen URLs und stößt die Analyse der jeweiligen Web Pages an. Dafür wird für jede URL ein `LanguageSeparator`-Objekt erzeugt.

Die Aufgabe des `LanguageSeparator`s ist es, die Verbindung zu der Web Page herzustellen, sämtlichen Code der Sprachen HTML, CSS und JavaScript zu extrahieren und diesen so vorzubereiten, dass er auf das Auftreten von Code Smells hin analysiert werden kann. Nach dem Zugriff auf den initialen HTML-Code (`rawHTML`) werden sämtliche CSS- und JavaScript-Code-Blöcke aus dem HTML-Code extrahiert. Zusätzlich werden sämtliche eingebundene CSS- und JavaScript-Dateien ebenfalls abgerufen. Es folgt die Erstellung eines ASTs für alle gefundenen JavaScript-Komponenten sowie vergleichbarer Code-Modelle für das bereinigte HTML und die kombinierten CSS-Komponenten. Da ein `LanguageSeparator`-Objekt unmittelbar an eine aktive Verbindung zu einer Website geknüpft ist, gibt es mit dem `LanguageSeparatorDummy` eine erbende Klasse, die auf diesen Aspekt verzichtet und das Testen des Codes deutlich vereinfacht und beschleunigt. Zusätzlich erhebt der `LanguageSeparator` für die differenzierten Analysen Daten darüber, wie viele Dateien, Code-Zeilen und Zeichen je Sprache aufgefunden wurden. `WebDeo` reicht diese Daten an die dritte Komponente, ein Objekt der Klasse `StatDocumentation` weiter.

Die Klasse `StatDocumentation` ist für das Speichern aller gesammelten Daten und des bisherigen Fortschritts zuständig. Mit dem Schreiben der Datei `currentState.json` wurde das Konzept des Web Crawlers zum Speichern des aktuellen Fortschritts, um das Unterbrechen und Fortführen der aktuellen Analyse zu ermöglichen, für *WebDeo* adaptiert. Über die Methoden `registerLanguageSeparatorStats()` und `registerDetectorStats()` kann `StatDocumentation` zwei Arten von Datensätzen entgegennehmen. Neben den zuvor beschriebenen Daten, die vom `LanguageSeparator` erhoben werden, handelt es sich dabei um die identifizierten Instanzen eines bestimmten Code Smells für eine bestimmte Website. Nachdem eine Website vollständig analysiert wurde, ruft `WebDeo` die Methoden `backupCurrentState()` und `backupData()` von `StatDocumentation` auf, wodurch sowohl der aktuelle Fortschritt (`currentState.json`) als auch alle Daten vom `LanguageSeparator` und den `CodeSmellDetection`-Objekten (`statDocumentation.json`) permanent geschrieben werden.

Als vierte Komponente wurde die Identifikation der Code Smells ausgemacht. Jeder der 39 Code Smells erhält dafür eine eigene Detektionsklasse. Die grundlegenden Gemeinsamkeiten werden in der Oberklasse `CodeSmellDetection` zusammengefasst. Da sich für JavaScript und CSS wiederholende Vorbereitungsschritte zur Erkennung der Smells ergeben haben, existiert für die beiden Sprachen eine weitere Oberklasse, in die diese Funktionalitäten ausgelagert wurden. Für HTML erwies sich dies als nicht notwendig, da das HTML-Modell von `BeautifulSoup` bereits alle notwendigen Funktionalitäten zur Identifikation direkt anbietet. Detektorklassen für

Smells, die sowohl in JavaScript als auch CSS identifiziert werden können (⇒ Beispiel `LenghtyLines` in Abbildung 6.2) erben sowohl von `JSCodeSmellDetection` als auch von `CSSCodeSmellDetection`.

In der Methode `mainProcedure` der Klasse `WebDeo` wird für jede URL nacheinander ein `LanguageSeparator` und je ein Objekt pro Code-Smell-Detektor erzeugt. Der Zugriff auf den Code in den Detektoren erfolgt durch Übergabe des `LanguageSeparator`-Objektes im Konstruktor. Über den `LanguageSeparator` erhalten die Detektionsklassen auch Zugriff auf die Daten aus `resources.json`, zum Beispiel auf für die Erkennung benötigte Listen, wie sie in Anhang A.2 zu sehen sind. Die Oberklasse `CodeSmellDetection` verfügt über eine Liste `smellOccurences`, in der die identifizierten Instanzen der jeweiligen Code Smells gespeichert werden. Dafür rufen die jeweiligen Unterklassen die gemeinsame Methode `recordSmellOccurence()` mit allen Informationen zu dem identifizierten Code Smell auf. Durch die Übergabe des Detektor-Objektes an die Methode `registerDetectorStats()` der Klasse `StatDocumentation` werden die Daten dauerhaft gespeichert.

6.3. WebDeo: Umsetzung

6.3.1. Test-First-Ansatz

Zum Sicherstellen der Funktionalität der Code-Smell-Erkennung wurde *WebDeo* nach dem Test-First-Ansatz entwickelt. Somit wurden zuerst Tests, welche die erwartete Funktionalität der Erkennung überprüfen können, und anschließend die Erkennung selbst entwickelt. Um die Erkennung über Unit-Tests auf die korrekte Funktionalität hin prüfen zu können, wird ein `LanguageSeparatorDummy`-Objekt erstellt. Dieses stellt keine tatsächliche Verbindung zu einer Website her, sondern wird mit Code-Modellen von HTML-, CSS- und JavaScript-Dateien erzeugt, in denen verschiedene Fälle und Ausprägungen der untersuchten Code Smells bewusst eingebaut worden sind. Dies ermöglicht es, mithilfe der Unit-Tests zu prüfen, ob die Code Smells anhand verschiedener Beispiele wie gewünscht identifiziert und ob die Schwellenwerte zur Identifikation korrekt angewendet werden.

```css
body {
    width: 40%;
    margin: 0 30%;
    padding: 10px;
    text-align: justify;
    font-family: Helvetica;
}
```

```css
body#tooLongRules {
    width: 40%;
    margin: 0 30%;
    padding: 10px;
    text-align: justify;
    font-family: Helvetica;
    text-decoration: underline;
}
```

Abbildung 6.3.: CSS-Code-Beispiel zum Testen des Schwellenwertes des Smells *Too Long Rules*

Abbildung 6.3 zeigt das Code-Beispiel in CSS zum Prüfen des Schwellenwertes für den Smell *Too long Rules*. Der zu prüfende Schwellenwert für die Metrik *Number of Values* liegt bei $NOV > 5$. Der Unit-Test prüft somit, ob der Smell in dem Beispiel auf der linken Seite mit $NOV = 5$ korrekterweise nicht identifiziert wird, in dem rechten Beispiel mit $NOV = 6$ jedoch schon.

6.3.2. Sprach-Separation und Modell-Erstellung

Sollte der `LanguageSeparator` keine erfolgreiche Verbindung zu einer Website herstellen können, wird diese umgehend aus der Analyse ausgeschlossen. Andernfalls erhält er initial Zugriff auf den HTML-Code und muss daraus den CSS- und JavaScript-Code extrahieren. Zur Einbindung von CSS- und JavaScript-Code gibt es mit externen Dateien und jeweiligen HTML-Tags (`<script>` und `<style>`) zwei grundlegende Möglichkeiten. Während der Code aus den Tags direkt extrahiert werden kann, müssen externe Dateien wiederum vom `LanguageSeparator` aufgerufen und ausgelesen werden.

Nachdem alle CSS- und JavaScript-Komponenten extrahiert wurden, werden diese zu jeweils einem vollständigen String kombiniert. Der HTML-Code wird wiederum um die aufgefundenen CSS- und JavaScript-Komponenten bereinigt. Somit ist die vollständige Separation der jeweiligen Sprachen erreicht. Es folgt die Erstellung des jeweiligen Code-Modells, das die statische Code-Analyse ermöglicht. Für HTML fiel die Wahl auf das weit verbreitete Web-Scraping-Tool `BeautifulSoup` [Ric]. Als CSS-Parser wurde sich für `tinycss2` [Sap] entschieden. Die statische Code-Analyse wird von diesem Parser mithilfe einiger vorbereitender Schritte, welche in die Oberklasse `CSSCodeSmellDetection` ausgelagert wurden, ermöglicht. Für den JavaScript-Code wird ein klassischer AST generiert. Die Wahl fiel dabei auf den Python-Port von `Esprima` [Kro] – ein Parser, der ursprünglich in JavaScript entwickelt wurde [Tea].

6.3.3. Framework-Filter

Bei ersten Testläufen des `LanguageSeparator`s offenbarte sich eine Herausforderung, die zuvor zwar bedacht, deren Ausmaß jedoch unterschätzt wurde. In der Web-Entwicklung werden häufig Frameworks eingesetzt, welche in vollem Umfang als externe CSS- oder JavaScript-Datei eingebunden werden. Diese Dateien enthalten somit sehr viel Code und das Parsen der Quellcode-Modelle erstreckte sich dadurch für einige Websites über mehrere Minuten.

Neben den Perfomance-Problemen stellten Frameworks ein weiteres Problem dar, welches der Duplikat-Problematik, welche im Rahmen von E1 in Kapitel 6.1 erklärt wurde, ähnelt. Frameworks wie *JQuery* oder *Bootstrap* sind so stark verbreitet, dass sie sehr häufig analysiert werden würden und somit massiven Einfluss auf die Ergebnisse der Analyse hätten. Der Umfang dieser Frameworks verstärkt diese Problematik zusätzlich.

Mit der großen Stichprobengröße für die geplante Analyse wird außerdem das Ziel verfolgt, die durchschnittliche Code-Qualität im Web möglichst gut abzubilden und so

ein realistisches Bild der Verbreitung von Code Smells zu erhalten. Durch ihre starke Verbreitung würden Frameworks auch zur Verfehlung dieses Ziels beitragen, da sie ausschließlich von professionellen Unternehmen entwickelt werden und daher von einer überdurchschnittlich hohen Code-Qualität ausgegangen werden kann.

Abschließend resultieren aus der Analyse von Frameworks Probleme hinsichtlich der Erkennung einiger Code Smells. Um die Dateigröße möglichst klein zu halten, werden Frameworks oft in einer `.min`-Version eingebunden. In diesen Dateien wird der Code nach der Entwicklung maximal komprimiert und von Leerzeichen, Zeilenumbrüchen und vergleichbaren Fortmatierungskomponenten befreit. Smells wie *Lengthy Lines* oder *Long Method* wären somit nicht mehr sinnvoll identifizierbar.

Herausforderung H4 ⇒ Entscheidung E4

Die wiederkehrende Analyse von Frameworks führt zu zahlreichen Problemen hinsichtlich der Performance, der Daten-Qualität und der korrekten Erkennung einiger Code Smells.

Aufgrund der vielen Problematiken erscheint es nicht nur vertretbar, sondern sinnvoll, Frameworks aus der Analyse auszuschließen.

Aufgrund der genannten Probleme wird die Entscheidung (E4) gefällt, weit verbreitete Frameworks aus der Analyse auszuschließen. Für die Umsetzung wird ein Filter in den `LanguageSeparator` eingebaut, der bestimmte CSS- und JavaScript-Dateien ignoriert:

- Dateien, die auf eine einzelne Zeile herunterkomprimiert wurden, dabei aber trotzdem über 10.000 Zeichen lang sind
- Dateien, die mehr als 1.000.000 Zeichen lang sind
- Dateien, deren Pfad bestimmte Strings enthält

```
"cssFileFilter":[
  "bootstrap",
  "dashicons",
  "elementor",
  "fontawesome",
  "font-awesome",
  "fontello.css",
  "ionicons",
  "jetpack",
  "jquery",
  "mediaelementplayer",
```

```
"javaScriptFileFilter":[
  "chunk.js",
  ".min.js",
  "Manifest.js",
  "adManager",
  "adsbygoogle",
  "analytics.js",
  "angular",
  "bootstrap",
  "bundle.js",
  "buttons.js",
```

Abbildung 6.4.: Ausschnitt der Liste an Strings, die als Frameworkfilter in CSS (links) und JavaScript (rechts) dienen

Die Liste an Strings für den Dateipfad-Filter wurden durch eine eigenständige Analyse zusammengetragen. Durch eine Modifizierung des Web Crawlers wurden

abermals 10.000 zufällige Websites analysiert und ausschließlich der Pfad von extern verlinkten JavaScript- und CSS-Dateien gespeichert. Der erhaltene Datensatz wurde auf wiederkehrende Strings hin untersucht. Sich häufig wiederholende Schlüsselwörter, die nach menschlicher Einschätzung auf Frameworks hinweisen, wurden in der Datei `resources.json` zusammengetragen und werden dem Filter im `LanguageSeparator` somit zur Verfügung gestellt. Ein Ausschnitt aus dieser Liste ist in Abbildung 6.4 zu sehen.

6.3.4. Erkennung der Code Smells

Bezüglich der Erkennung der Code Smells ist es gelungen, für alle 39 untersuchten Smells einen funktionierenden Detektor, unter der Benutzung der zuvor ermittelten Metriken und Schwellenwerte, zu entwickeln. Auf einige besondere Herausforderungen und Erkenntnisse wird im Folgenden näher eingegangen.

Bezüglich der Identifikation von JavaScript-Smells offenbarte sich schnell eine große Herausforderung durch eine hohe Komplexität. Der AST von analysierten Websites besteht aus großen und tief verschachtelten Listen und Dictionaries. Um dennoch den gesamten AST zu durchlaufen, ohne Pfade zu verpassen, erwies sich die Nutzung von Rekursion als einzig sinnvolle Lösung. Da während des Rekursionsprozesses Daten als Anhaltspunkte für potenzielle Code Smells identifiziert, gespeichert, weitergereicht, erhöht und an den korrekten Stellen auch wieder zurückgesetzt werden mussten, entwickelte sich die Erkennung einiger JavaScript-Smells zur logisch komplexesten Aufgabe des gesamten Projektes.

```
1  # Traverses the whole JavaScript AST including Nested Nodes
2  # via Recursion and checks each node for the Code Smell of
3  # the smellObj
4  def traverseAST(self, smellObj, node, metaData):
5      if node != None:
6          smellObj.astDetection(node, metaData)
7          for key in node:
8              child = node[key]
9              if isinstance(child, list):
10                 for childNode in child:
11                     self.traverseAST(smellObj, childNode, metaData)
12             else:
13                 if isinstance(child,dict):
14                     self.traverseAST(smellObj, child, metaData)
```

Abbildung 6.5.: Rekursionsalgorithmus zum vollständigen Analysieren des ASTs

Als einer der absoluten Schlüsselpunkte des Codes wird der Rekursionsalgorithmus `traverseAST` aus der Oberklasse `JSCodeSmellDetection` in Abbildung 6.5 dargestellt. Die Methode enthält drei Parameter. Bei `smellObj` handelt es sich um das Objekt der jeweiligen Detektoren-Unterklasse, zum Beispiel ein Objekt der Klasse `ArgumentCountMismatch`. `node` ist der spezifische Knoten des ASTs, der bei dem je-

weiligen Methodenaufruf analysiert werden soll. Mit `metaData` ist es möglich, verschiedene Werte, über die Rekursionsprozedur hinweg, in Form eines Dictionaries weiterzureichen und zu verändern.

Jede der von `JSCodeSmellDetection` erbenden Detektorenklassen enthält eine eigene Umsetzung der Methode `astDetection()`, welche aufgerufen wird (⇒ Code-Zeile 6) und erkennt, ob in dem übergebenen Knoten ein Smell vorliegt. Ist dies der Fall, wird die identifizierte Smell-Instanz auch direkt gespeichert. Anschließend werden alle Schlüssel-Wert-Paare des Knotens abgelaufen. Sollte es sich bei ihnen um primitive Datentypen oder Zeichenketten handeln, sind keine weiteren Schritte notwendig. Im Fall von Dictionaries oder Listen liegt jedoch eine Verschachtelung weiterer Knoten vor. Daher erfolgt im Fall von Dictionaries (⇒ Code-Zeile 13) ein rekursiver Aufruf von `astDetection()` für den Kinderknoten (⇒ Code-Zeile 14), und im Falle von Listen (⇒ Code-Zeile 9) sogar je ein rekursiver Aufruf je Element in dem Listen-Kinderknoten (⇒ Code-Zeilen 10-11).

In dieser Form des Algorithmus wird `metaData` genutzt, um mehrere Durchläufe durch den AST differenzieren zu können. Für den Smell *Argument Count Mismatch* ist es zum Beispiel notwendig, den AST zweimal zu durchlaufen. Im ersten Durchlauf werden alle Methoden-Deklarationen und die Länge ihrer Parameterliste gespeichert. Im zweiten Durchlauf werden alle Methoden-Aufrufe auf Abweichungen von der Deklaration geprüft.

Für die Erkennung einiger Code Smells war diese Grundform des Algorithmus nicht ausreichend. In diesem Fall wird die Methode `traverseAST()` in den erbenden Detektorenklassen überschrieben und um weitere Funktionalitäten ergänzt. Folgende Fälle traten dabei in den gelisteten Klassen auf, die die Komplexität des rekursiven Algorithmus noch deutlich erhöhten:

- Ein Stopp-Signal für die Rekursion wurde als Rückgabewert benötigt, damit eine Smell-Instanz bei deutlicher Überschreitung des Schwellenwertes nicht mehrfach gezählt wird: `ChainedMethods`, `Depth`, `NestedCallbacks`
- Ein Zähler für die Metrik musste ergänzt werden und gegebenenfalls vor dem rekursiven Aufruf erhöht oder zurückgesetzt werden: `Depth`, `NestedCallbacks`
- Zur korrekten Identifikation von globalen Variablen musste für jeden analysierten Knoten bekannt sein, ob er sich innerhalb eines Block- oder Function-Scopes [w3sf] befindet: `ExcessiveGlobalVariables`

Durch das Auslagern der `astDetection`-Methode in die jeweiligen Detektorenklassen ist es erfolgreich gelungen, die Komplexität der Rekursion zu beherrschen und den in Abbildung 6.5 dargestellten Algorithmus übersichtlich zu halten. Die logische Herausforderung, den Rekursionsalgorithmus mit der eigentlichen Code-Smell-Erkennung zu kombinieren, erwies sich hingegen häufig als sehr kompliziert. Allgemein wurde während der Implementation deutlich, dass die Identifikation jedes einzelnen Code Smells,

aufgrund ihrer hohen Unterschiedlichkeit, eine sehr individuelle Herausforderung darstellt. Nur selten konnte bei der Implementation eines weiteren Detektors auf bereits zuvor verwendete, logische Schritte zurückgegriffen werden.

Auch die Grenzen der statischen Code-Analyse wurden während der Implementation der JavaScript-Detektoren deutlich. Um Code Smells innerhalb des ASTs zu identifizieren, müssen dem Detektor genaue Schablonen vorgegeben werden, welche beschreiben, nach welchen Mustern im Code gesucht werden muss. Da die Syntax von Programmiersprachen jedoch derartig vielseitig und abwechslungsreich ist, ist es Entwickler*innen kaum möglich, alle Eventualitäten in diesen Schablonen abzudecken.

Ein Beispiel hierfür bietet der Smell *Array Length Assignment*. Auf den ersten Blick wirkt der Code Smell sehr simpel. Es muss lediglich erkannt werden, ob das Attribut `length` einer Array-Variable verändert wird. Bei der Umsetzung der Identifikation gestaltet sich dies jedoch äußerst schwierig. Da Variablen in JavaScript nicht typgebunden sind, können Typen auch nicht ohne weiteres abgefragt werden. Um Array-Variablen identifizieren zu können, ist bereits ein vollständiger Durchlauf des ASTs notwendig, bei dem Schablonen für *jede erdenkliche Art* an ein Array zu gelangen, benötigt werden. Die Komplexität dieses Problems wird schnell deutlich:

- Es müssen Variablenzuweisungen gesucht werden, auf deren rechter Seite sich eine `ArrayExpression` befindet (`[]`).
- Es müssen Variablenzuweisungen gesucht werden, auf deren rechter Seite sich eine `NewExpression` befindet. Auf diese `NewExpression` muss an einem bestimmten Pfad des ASTs das Stichwort `Array` folgen (`new Array()`).
  - Eine solche Schablone ist für *jede* erbende Klasse von `Array` notwendig.
    - Dies inkludiert sämtliche erbenden Klassen der offiziellen JavaScript-Syntax.
    - Dies inkludiert sämtliche eigenen Klassen, welche von `Array` erben.
      - Für deren Identifikation wären wiederum unzählige Schablonen notwendig.
- Es müssen Variablenzuweisungen gesucht werden, auf deren rechter Seite sich eine `CallExpression` befindet, die ein `Array` zurückgibt.
  - Der Rückgabetyp einer Methode müsste über unzählige weitere Schablonen analysiert werden.
  - Statt einem direkten Methoden-Aufruf (`FunctionExpression` in `CallExpression`) könnte sich auf der rechten Seite auch eine Aneinaderkettung von Methodenaufrufen befinden.
    - Diese Länge dieser Methodenkette ist komplett variabel.
    - Statt `FunctionExpression`s könnten sich in der Aufrufkette auch Objekte befinden. Auch diese könnten sich an jeder beliebigen Stelle der Aufrufkette befinden.
- ...

Die Implementation von Code-Smell-Detektoren ähnelt aufgrund dieses Umstands einem Hinterherjagen der Möglichkeiten der Programmiersprache. Mit den Mitteln der statischen Code-Analyse kann man daher kaum eine sehr gute Identifikationsquote erreichen. Nicht grundlos wird aktiv an weiteren Erkennungsstrategien für Code Smells geforscht (⇒ Kapitel 2.2.3). Es ist jedoch davon auszugehen, dass man mit überschaubarem Aufwand und der Identifikation der üblichsten Umsetzungen im Code auch bereits einen großen Anteil an Smells auffinden dürfte, wenngleich seltene Fälle nicht identifiziert werden können. Der Test-First-Ansatz (⇒ Kapitel 6.3.1) half dabei, die verschiedenen Fälle des Auftretens der Smells bei der Erkennung zu berücksichtigen. Dennoch handelt es sich auch bei den Testfällen lediglich um die Fälle, die einem einzelnen Entwickler bekannt waren.

Für CSS und insbesondere HTML erwies sich die Detektorenimplementation aufgrund der weniger vielseitigen Syntax als deutlich einfacher. Besonders hervorzuheben ist jedoch die in *WebDeo* implementierte Erkennung des HTML-Smells *Opening Tag missing Closing Tag*. Nachdem der Versuch der Identifikation über reguläre Ausdrücke unzufriedene Resultate lieferte, wurde sich mit den verschiedenen HTML-Parsern, die in Python angewendet werden können, beschäftigt. Nachdem auch keiner dieser Parser das gewünschte Resultat lieferte, wurde ein eigener `TagDetectingHTMLParser` als Teil von *WebDeo* entwickelt, der exakt die gewünschte Funktionalität anbietet.

6.4. Feedback-Bericht

Auf die Implementation der Code-Smell-Erkennung folgte die Durchführung der geplanten Verbreitungsanalyse. Durch erste Testläufe und die Analyse der Output-Daten konnten letzte Fehler im Programmablauf und bei der Identifikation der Smells ausgemacht und ausgebessert werden. Die zusammengetragenen Daten wurden aus der `statDocumentation.json`-Datei in Microsoft Excel übertragen und darin ausgewertet. Die Ergebnisse werden im Rahmen von Kapitel 7 vorgestellt. Teile der Ergebnisse wurden bereits zu diesem Zeitpunkt der Entwicklung benötigt, da sie Bestandteil des geplanten Feedback-Berichtes sind.

Im nächsten Schritt der Entwicklung sollte *WebDeo* um die Möglichkeit ergänzt werden, ebendiesen Feedback-Bericht zu generieren. In dem Bericht sollten alle identifizierten Code Smells optisch aufbereitet präsentiert werden, um Entwickler*innen dabei zu unterstützen, diese gezielt aus ihrem Code zu entfernen. Zusätzlich sollten persönliche Stärken und Schwächen in der Code-Qualität der Nutzer*innen ermittelt und präsentiert werden. Dafür sollte die analysierte Verbreitung der Code Smells im Code der Nutzer*innen mit den ermittelten Durchschnittswerten aus dem World Wide Web abgeglichen werden. Die Resultate des Abgleichs sollten, ebenfalls optisch aufbereitet, in dem Bericht präsentiert werden.

Die Entscheidung für das Format des Feedback-Berichts fiel auf HTML-Dateien. Im

Gegensatz zum Generieren von PDF-Dateien lassen sich dabei einige wünschenswerte Features deutlich simpler umsetzen:

- Nutzung von HTML-Templates zum Generieren des Berichtes
- Nutzung von HTML-Features, wie das Setzen von Links
- Ansprechende und platzsparende Designs durch CSS
- Hinzufügen von Funktionalität über JavaScript

Für die Umsetzung des Feedback-Berichts wurde *WebDeo* um eine weitere Klasse, `ReportGenerator`, ergänzt. Ihre Ergänzung in die vorherige System-Architektur (⇒ Abbildung 6.2) ist Abbildung 6.6 zu entnehmen.

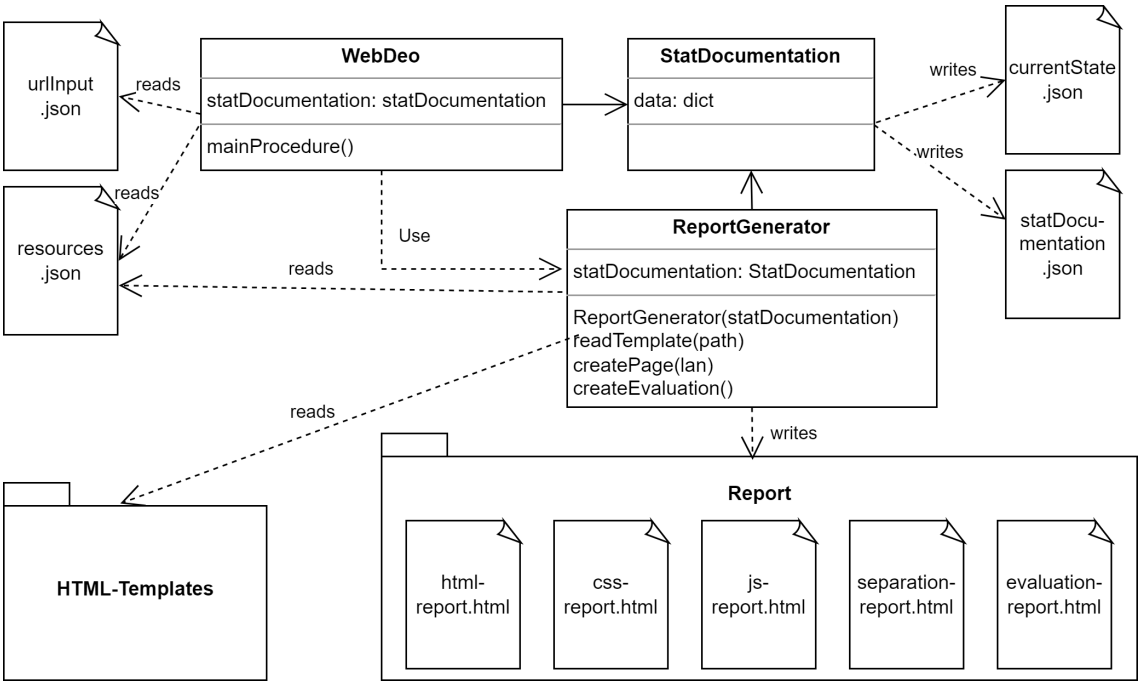


Abbildung 6.6.: Eingliederung des `ReportGenerator`s in die System-Architektur (zugunsten der Übersichtlichkeit verkürzt)

Alle Komponenten des Feedback-Berichtes wurden vorab in HTML und CSS designed. Anschließend wurden die sich wiederholenden Elemente sowie der grundlegende Aufbau des Berichtes in sieben HTML-Templatedateien unterteilt. Variable Werte und Textstellen wurden mithilfe von Platzhaltern in die Templates gesetzt, welche beim Generieren mit dem entsprechenden Inhalt formatiert werden.

Neben den HTML-Templates erhält der `ReportGenerator` zwei weitere wesentliche Inputs. Alle benötigten statischen Daten wurden der Datei `resources.json` hinzugefügt, welche von `ReportGenerator` eingelesen wird. Die Daten der aktuellen Analyse, auf deren Basis der Bericht generiert wird, erhält der `ReportGenerator` hingegen als Parameter. Hierfür übergibt die Hauptklasse `WebDeo` am Ende der aktuellen Analyse das `StatDocumentation`-Objekt, welches sämtliche zusammengetragenen Daten enthält, bei der Erzeugung an den `ReportGenerator`.

Unter Einsatz einiger Hilfsmethoden, welche aus Gründen der Übersichtlichkeit in Abbildung 6.6 nicht dargestellt sind, füllt der `ReportGenerator` anschließend die eingelesenen HTML-Templates und setzt somit fünf Output-Dateien zusammen. Mithilfe der Methode `createPage(lan)` wird für die vier Smell-Kategorien (HTML, CSS, JavaScript und Separation) je ein Bericht mit allen identifizierten Code Smells generiert. Die `createEvaluation()`-Methode erstellt zusätzlich die Datei `evaluation-report.html`, in der die ermittelte Code-Qualität mit den Durchschnittswerten abgeglichen wird. Wenngleich fünf separate Report-Dateien generiert werden, ergeben diese, aufgrund des übereinstimmenden Designs und da die einzelnen Seiten aufeinander verlinken, ein einheitliches Gesamtbild. Zusätzlich wird der Bericht mit dem Ende des Generiervorgangs automatisch im Browser geöffnet, wodurch Nutzer*innen nicht zu den Dateien navigieren müssen.

Abbildung 6.7.: Navigation im Feedback-Bericht

Die vier Smell-Berichte beginnen mit einer Navigation, in der alle untersuchten Smells der jeweiligen Kategorie und die Anzahl identifizierter Instanzen aufgelistet werden (⇒ Abbildung 6.7). Durch einen Klick auf den jeweiligen Navigationseintrag können Nutzer*innen zu dem Abschnitt des jeweiligen Smells springen.

In den Abschnitten zu den einzelnen Code Smells wird eine Vielzahl an Informationen präsentiert (⇒ Abbildung 6.8). Neben einer kurzen Beschreibung des Smells handelt es sich dabei um die Anzahl aufgefundener Instanzen sowie empfohlene Refactorings, inklusive einer Verlinkung auf weitere Details zu deren Umsetzung.

Es folgt eine scrollbare Auflistung aller identifizierten Instanzen des jeweiligen Smells. Dabei werden die jeweiligen Stellen im Code präsentiert, an denen der Smell identifiziert wurde. Für ausgewählte Smells wird die entscheidende Stelle innerhalb der angezeigten Code-Umgebung zusätzlich gelb hervorgehoben, um die Nachvollziehbarkeit zu erhöhen. Je nach Smell folgen bis zu zwei kurze Textparagraphen mit weiteren Informationen dazu, welche Ausprägung des Smells konkret vorliegt. Zusätzlich wird die Web Page, auf der die jeweilige Smell-Instanz identifiziert wurde, angegeben, da ein Feedback-Bericht

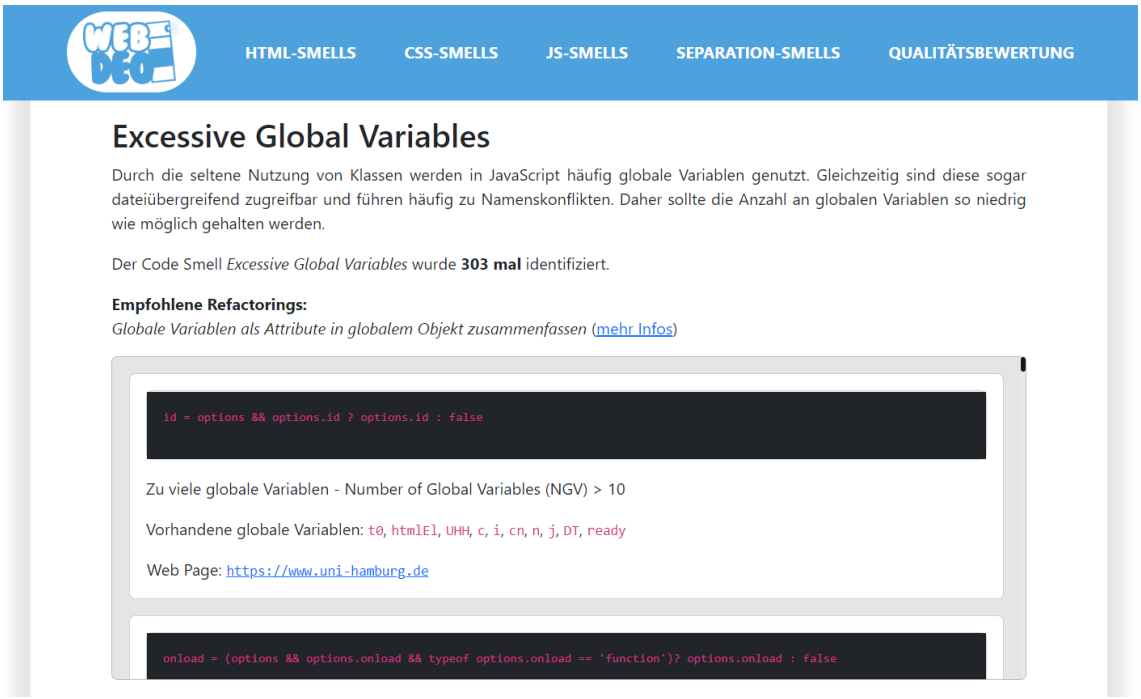


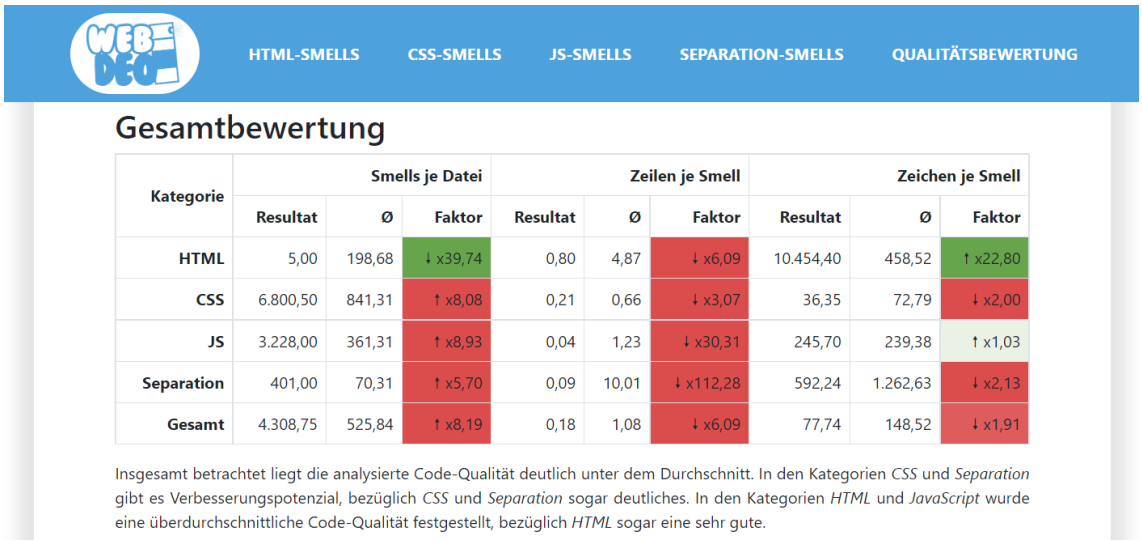
Abbildung 6.8.: Code-Smell-Abschnitt im Feedback-Bericht

auch für mehrere Web Pages gleichzeitig erstellt werden kann. Bei der Implementation dieses Features wurde sehr davon profitiert, dass bereits bei der Entwicklung der jeweiligen Code-Smell-Erkennungen eine Lokalisierung des Codes sowie die Sammlung zusätzlicher Details bedacht wurde. Somit lagen bereits alle benötigten Daten vor und diese mussten lediglich optisch aufbereitet werden.

Mit `evaluation-report.html` wird eine fünfte Berichts-Datei generiert, welche sich von den anderen Dateien deutlich unterscheidet. Anstatt konkrete Smell-Instanzen aufzulisten, wird die Code-Qualität in diesem Teil des Berichts ermittelt, präsentiert und mit den im Rahmen der durchgeführten Analyse ermittelten Durchschnittswerten im World Wide Web abgeglichen.

Um den Vergleich ziehen zu können, wurden drei verschiedene Durchschnittskennwerte für die Häufigkeit der Code Smells im analysierten Code aufgestellt: *Smells per File* (SpF), *Lines per Smell* (LpS) und *Characters per Smell* (CpS). Diese Werte wurden aus dem Datensatz zur durchschnittlichen Verbreitung der Smells im World Wide Web errechnet und dem `ReportGenerator` über die Datei `resources.json` zur Verfügung gestellt. Außerdem berechnet der `ReportGenerator` die gleichen Kennzahlen für eine neue Analyse mit Bericht-Erstellung. Der eigentliche Vergleich zwischen dem Durchschnittswert und dem Wert der aktuellen Analyse wird durch die Berechnung des Faktors zwischen den beiden Werten ermöglicht. Im Beispiel von Abbildung 6.9 liegt der ermittelte *CpS*-Wert für alle CSS-Smells bei 36,35 und im Durchschnitt bei 72,79. Für den Faktor X ergibt sich aus der Rechnung $36,35 \cdot X = 72,79$ der Wert $X \approx 2,00$.

Durch unterschiedliche Formatierungen, Dateigrößen und Zeilenlängen kann es po-



HTML-SMELLS CSS-SMELLS JS-SMELLS SEPARATION-SMELLS QUALITÄTSBEWERTUNG

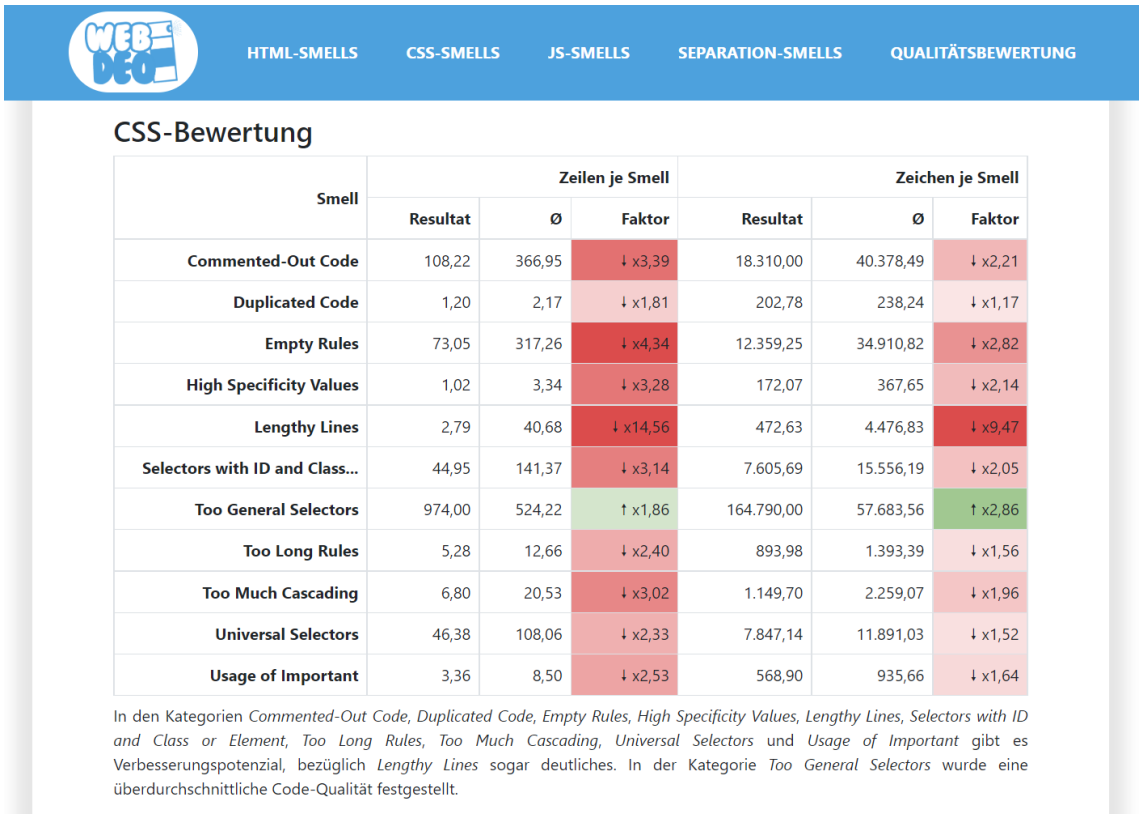
Gesamtbewertung

Kategorie	Smells je Datei			Zeilen je Smell			Zeichen je Smell		
	Resultat	Ø	Faktor	Resultat	Ø	Faktor	Resultat	Ø	Faktor
HTML	5,00	198,68	↓ x39,74	0,80	4,87	↓ x6,09	10.454,40	458,52	↑ x22,80
CSS	6.800,50	841,31	↑ x8,08	0,21	0,66	↓ x3,07	36,35	72,79	↓ x2,00
JS	3.228,00	361,31	↑ x8,93	0,04	1,23	↓ x30,31	245,70	239,38	↑ x1,03
Separation	401,00	70,31	↑ x5,70	0,09	10,01	↓ x112,28	592,24	1.262,63	↓ x2,13
Gesamt	4.308,75	525,84	↑ x8,19	0,18	1,08	↓ x6,09	77,74	148,52	↓ x1,91

Insgesamt betrachtet liegt die analysierte Code-Qualität deutlich unter dem Durchschnitt. In den Kategorien *CSS* und *Separation* gibt es Verbesserungspotenzial, bezüglich *CSS* und *Separation* sogar deutliches. In den Kategorien *HTML* und *JavaScript* wurde eine überdurchschnittliche Code-Qualität festgestellt, bezüglich *HTML* sogar eine sehr gute.

Abbildung 6.9.: Gesamtbewertung im Feedback-Bericht

tentiell zu starken Abweichungen zwischen den ermittelten Faktoren der verschiedenen Kennwerte kommen (⇒ Abbildung 6.9, Zeile *HTML*). Da die Anzahl der Zeichen im *CpS*-Wert die kleinste ermittelte Einheit darstellt, wird der Wert am wenigsten von solchen Effekten beeinflusst und ist am aussagekräftigsten. Daher wurde der Bericht auch um die Möglichkeit ergänzt, die *SpF*- und *LpS*-Werte auszublenden (⇒ Abbildung 6.10). Zusätz-



HTML-SMELLS CSS-SMELLS JS-SMELLS SEPARATION-SMELLS QUALITÄTSBEWERTUNG

CSS-Bewertung

Smell	Zeilen je Smell			Zeichen je Smell		
	Resultat	Ø	Faktor	Resultat	Ø	Faktor
Commented-Out Code	108,22	366,95	↓ x3,39	18.310,00	40.378,49	↓ x2,21
Duplicated Code	1,20	2,17	↓ x1,81	202,78	238,24	↓ x1,17
Empty Rules	73,05	317,26	↓ x4,34	12.359,25	34.910,82	↓ x2,82
High Specificity Values	1,02	3,34	↓ x3,28	172,07	367,65	↓ x2,14
Lengthy Lines	2,79	40,68	↓ x14,56	472,63	4.476,83	↓ x9,47
Selectors with ID and Class...	44,95	141,37	↓ x3,14	7.605,69	15.556,19	↓ x2,05
Too General Selectors	974,00	524,22	↑ x1,86	164.790,00	57.683,56	↑ x2,86
Too Long Rules	5,28	12,66	↓ x2,40	893,98	1.393,39	↓ x1,56
Too Much Cascading	6,80	20,53	↓ x3,02	1.149,70	2.259,07	↓ x1,96
Universal Selectors	46,38	108,06	↓ x2,33	7.847,14	11.891,03	↓ x1,52
Usage of Important	3,36	8,50	↓ x2,53	568,90	935,66	↓ x1,64

In den Kategorien *Commented-Out Code*, *Duplicated Code*, *Empty Rules*, *High Specificity Values*, *Lengthy Lines*, *Selectors with ID and Class or Element*, *Too Long Rules*, *Too Much Cascading*, *Universal Selectors* und *Usage of Important* gibt es Verbesserungspotenzial, bezüglich *Lengthy Lines* sogar deutliches. In der Kategorie *Too General Selectors* wurde eine überdurchschnittliche Code-Qualität festgestellt.

Abbildung 6.10.: Bereichsbewertung im Feedback-Bericht (*SpF*-Wert ausgeblendet)

lich wird aus den wichtigsten Erkenntnissen des Vergleichs ein Text generiert, welcher sich aus den genannten Gründen ebenfalls auf den *CpS*-Wert stützt.

Neben einer Gesamtbewertung je Kategorie (⇒ Abbildung 6.9) verfügt der Bericht auch über einen Vergleichsabschnitt je Kategorie für die einzelnen Smells (⇒ Abbildung 6.10). In diesem Abschnitt werden die Kennwerte für alle einzelnen Smells ermittelt, damit Nutzer*innen erfahren können, welche Smells sie besonders häufig einbauen. Dieses Feedback soll Entwickler*innen gezielt dabei unterstützen, an ihren Schwächen arbeiten zu können und das Bewusstsein bezüglich persönlicher Stärken zu verbessern. Neben der Auflistung der ermittelten Kennwerte, werden auch diese Erkenntnisse in einem Text hervorgehoben. Zusätzlich wird, je nach Wert des Faktors, ein unterschiedlicher Farbwert berechnet, in dem die jeweilige Tabellenzelle eingefärbt wird. Besonders große Schwächen oder Stärken werden somit auch optisch hervorgehoben und sind leicht zu identifizieren (⇒ Abbildung 6.10).

7. Ergebnisse

Ergebnis RQ.I)

Die Ergebnisse zu RQ.I) inklusive der Teilfragen RQ.Ia), RQ.Ib) und RQ.Ic) sind bereits Kapitel 5 zu entnehmen.

Ergebnis: Tool zur Identifikation von Code Smells

Mit *WebDeo* wurde, als weiteres Ergebnis dieser Arbeit, ein Tool entwickelt, welches Code Smells in den Sprachen HTML, CSS und JavaScript sowie sprachübergreifende *SoC*-Smells identifiziert. Das Tool wurde bereits in den Unterkapiteln 6.2 – 6.4 präsentiert.

7.1. Analysierte Stichprobe

Im Rahmen des Kapitels 7 werden die Ergebnisse der durchgeführten Analyse präsentiert und interpretiert. Sämtliche Dezimalwerte werden dabei auf zwei Nachkommastellen gerundet. Um eindeutige und gleichzeitig pragmatische Abkürzungen bezüglich der Kennwerte zu erhalten, wurde sich für englische Bezeichner und Kürzel entschieden. So fällt zum Beispiel die Unterscheidung zwischen Zeilen und Zeichen (jeweils *Z* in deutscher Sprache) in der englischen Sprache leichter (*L (Lines)* und *C (Characters)*).

Um die Ergebnisse der durchgeführten Analyse möglichst differenziert betrachten zu können, wurde für die Stichprobe neben der Website-Anzahl zusätzlich die Anzahl analysierter Dateien, Zeilen und Zeichen ermittelt. Die genauen Werte für das Ausmaß der Stichprobe sind Tabelle 7.1 zu entnehmen. Aus insgesamt 10.255 analysierten Websites – repräsentiert durch je eine Web Page – wurden 65.089 unterschiedliche Dateien separiert und auf vorliegende Code Smells untersucht. Dies entspricht wiederum fast 37 Mio. Zeilen an analysiertem Code und mehr als 5 Mrd. untersuchten Zeichen.

Durch die Analyse von 10.255 Websites wurde das Ziel einer umfangreichen und aussagekräftigen Stichprobe von mindestens 10.000 Websites erreicht. In vergleichbaren Arbeiten lag die Anzahl untersuchter Websites durchschnittlich bei 803,5, bei einem Median von 13 und maximal bei 5.525 (⇒ Tabelle 3.1). Somit wurden im Rahmen dieser Arbeit deutlich mehr Websites als in den vorgestellten vergleichbaren Arbeiten analysiert. Gemäß des „Gesetzes der großen Zahlen“ kann somit auch von aussagekräftigeren Ergebnissen ausgegangen werden, insbesondere bezüglich ermittelter Durchschnittswerte.

Um die Code Smells, welche Verstöße gegen das Prinzip *Separation of Concerns* darstellen, im weiteren Verlauf der Ergebnisinterpretation mit den anderen Smells vergleichen zu können, wurden die in Tabelle 7.1 gelisteten Werte berechnet. Da diese Smells

Kategorie	Websites	Dateien	Zeilen	Zeichen
HTML	10.255	10.255	9.923.574	934.210.320
CSS		23.512	13.085.563	1.439.896.905
JavaScript		31.322	13.934.668	2.709.042.117
SoC*		15.522	10.926.348	1.377.918.269
Gesamt	10.255	65.089	36.943.805	5.083.149.342

Tabelle 7.1.: Absolute Werte der analysierten Stichprobe

* Die SoC-Werte ergeben sich rechnerisch und zählen nicht zur Zeile *Gesamt* (siehe Erläuterung im Text)

in verschiedenen Sprachen auftreten, wurden die Werte ihrer jeweiligen Sprache anteilig gewichtet. Daraus resultieren Werte, welche die Berechnungen, die für die drei Sprachen vorgenommen werden, auch für die Kategorie *SoC* ermöglichen. Diese Werte tragen nicht zu dem Umfang der Stichprobe – und somit auch nicht zu den Werten der Zeile *Gesamt* – bei. Insgesamt wurden acht *SoC*-Smells untersucht, von denen zwei in JavaScript und sechs in HTML identifiziert wurden (⇒ Kapitel 5.6). Die Berechnung der *SoC*-Werte ergab sich daher wie folgt:

$$Wert(SoC) = \frac{2 \cdot Wert(JS) + 6 \cdot Wert(HTML)}{8}$$

Aus den in Tabelle 7.1 dargestellten Werten können unterschiedliche Durchschnittswerte ermittelt werden (⇒ Tabelle 7.2), die wichtige Hinweise zur korrekten Deutung der Ergebnisse liefern.

Kategorie	FpW	LpW	CpW	LpF	CpF	CpL
HTML	1,00	967,68	91.098,03	967,68	91.098,03	94,14
CSS	2,29	1.276,02	140.409,25	556,55	61.240,94	110,04
JavaScript	3,05	1.358,82	264.167,93	444,88	86.490,07	194,41
SoC*	1,51	1.065,47	134.365,51	703,94	88.773,38	126,11
Gesamt	6,35	3.602,52	495.675,22	567,59	78.095,37	137,59

Tabelle 7.2.: Berechnete Durchschnittswerte der analysierten Stichprobe

* Die SoC-Werte ergeben sich rechnerisch und zählen nicht zur Zeile *Gesamt* (siehe Erläuterung im Text)
FpW: Files per Website, LpW: Lines per Website, CpW: Characters per Website, CpF: Characters per File, CpL: Characters per Line

Aus den in Tabelle 7.2 dargestellten Werten wird deutlich, dass der Code-Umfang je Website, Datei und Zeile deutlich zwischen den verschiedenen analysierten Sprachen variiert. So enthält zum Beispiel eine durchschnittliche Zeile in JavaScript (194,41) mehr als doppelt so viel Code (gemessen an der Zeichenanzahl) wie eine durchschnittliche

HTML-Zeile (94,14). Gleichzeitig besteht eine einzelne HTML-Datei jedoch durchschnittlich aus mehr als doppelt so vielen Zeilen (967,68) wie eine JavaScript-Datei (444,88).

Ursache für diese teils deutlichen Unterschiede sind, neben der sehr unterschiedlichen Syntax der Sprachen, insbesondere die verschiedenen Formatierungsarten. Unabhängig von den unterschiedlichen Formatierungen gilt jedoch: Je mehr Zeichen der Code enthält, desto mehr Code Smells können sich auch potenziell in dem Code befinden. Wie im Rahmen von Kapitel 6.4 bereits erwähnt wurde, handelt es sich daher bei den Kennwerten, die sich auf die analysierte Zeichenanzahl beziehen, wohl um die am besten vergleichbaren Ergebnisse, da es sich um die feinste ermittelte Einheit handelt.

Die in Tabelle 7.2 dargestellten Unterschiede zwischen den Durchschnittswerten lassen sich auf weitere Ergebnisse übertragen. Sie sollten daher bei der Deutung aller folgenden *per Website*-, *per File*- und *per Line*-Ergebnisse stets bedacht werden.

7.2. Allgemeine Verbreitung von Code Smells im Web

Tabelle 7.3 zeigt wie viele Code Smells je Kategorie untersucht wurden (NoAS) und wie viele Instanzen dieser Smells in der gesamten Stichprobe identifiziert wurden (ISI). Die Gesamtanzahl von 45 untersuchten Code Smells variiert in dieser Auflistung von der Anzahl 39 unterschiedlicher untersuchter Code Smells, da einige Smells in mehreren Sprachen identifiziert wurden (⇒ Kapitel 5.2).

Kategorie	NoAS	ISI	Anteil*
HTML	8	2.037.445	5,95%
CSS	12	19.780.781	57,79%
JavaScript	17	11.316.864	33,06%
SoC	8	1.091.308	3,19%
Gesamt	45	34.226.398	100,00%

Tabelle 7.3.: Anzahl untersuchter Smells und identifizierter Smell-Instanzen

* Im Anteil ist die unterschiedliche Anzahl untersuchter Smells (NoAS) nicht berücksichtigt
NoAS: Number of Analyzed Smells, ISI: Identified Smell Instances

Insgesamt wurden bei der durchgeführten Analyse 34.226.398 Instanzen der analysierten Code Smells identifiziert. Dabei gibt es deutliche Unterschiede zwischen den verschiedenen Kategorien. Die wenigsten Smell-Instanzen (1.091.308; 3,19%) wurden im Bereich der *SoC*-Smells festgestellt. Etwa doppelt so viele Smells (2.037.445; 5,95%) wurden in der Sprache HTML identifiziert. Deutlich häufiger wurden JavaScript-Smells (11.316.864; 33,06%) und insbesondere CSS-Smells (19.780.781; 57,79%) erkannt.

Im weiteren Verlauf dieses Kapitels werden die deutlichen Unterschiede dieser Werte durch das Einbeziehen des *NoAS*-Wertes und der in Tabelle 7.2 dargestellten Unterschie-

de in der Code-Formatierung noch differenzierter und aussagekräftiger betrachtet.

Da die Daten, welche Code Smells in welcher Sprache wie oft auftreten, für jede einzelne Website erhoben wurden, lässt sich die durchschnittliche Verbreitung der Code Smells je Website sehr differenziert betrachten (⇒ Tabelle 7.4).

Kat.	MIN	MAX	$\bar{x}$	CV	95%-Konfidenzintervall		$\frac{\bar{x}}{NoAS}$
					Untergrenze	Obergrenze	
HTML	0	18.116	198,68	211,21%	190,56	206,80	24,83
CSS	0	37.692	1.928,89	3,04%	1.927,76	1.930,02	160,74
JS	0	17.106	1.103,55	3,49%	1.102,80	1.104,29	64,91
SoC	0	6.308	106,42	12,43%	106,16	106,67	13,30
Alle	0	42.524	3.337,53	1,97%	3.336,26	3.338,81	74,17

Tabelle 7.4.: Verbreitung der Code Smells nach Kategorie je Website

MIN: Minimalwert, MAX: Maximalwert, $\bar{x}$: Mittelwert, CV: Variationskoeffizient, NoAS: Number of Analyzed Smells

Anhand der dargestellten MIN-Werte wird deutlich, dass sich für jede Kategorie Websites in dem Datensatz befanden, auf denen keiner der untersuchten Smells festgestellt werden konnte. Von den 10.255 Websites waren 6 sogar über alle Kategorien hinweg „smellfrei“ (0,0585%).

Bezüglich der ermittelten MAX-Werte zeigt sich, wie extrem einzelne Websites von Code Smells betroffen sind. So konnten auf einer einzelnen Website 42.524 Instanzen der analysierten Smells identifiziert werden. In Hinblick auf den höchsten Wert je Kategorie schneiden die *SoC*-Smells mit einem Wert von 6.308 noch am wenigsten schlecht ab. Für JavaScript (17.106) und HTML (18.116) ergaben sich relativ ähnliche Werte. Mit 37.692 identifizierten CSS-Smells auf einer einzelnen Website ist dieser Maximalwert, im Vergleich zu den HTML- und JavaScript-Werten, sogar mehr als doppelt so hoch.

Durchschnittlich konnten mit der Erkennung von *WebDeo* 3.337,53 Code Smells je Website identifiziert werden. In Anbetracht dessen, dass lediglich der eingebundene Code einzelner Web Pages und nicht die kompletten Websites analysiert wurden, ist dieser Mittelwert auffallend hoch. Bezüglich der einzelnen Kategorien gibt es auch hier ein starkes Gefälle. Mit einem Mittelwert von 106,42 und 198,68 Smells je Website sind die Werte für *SoC*- und HTML-Smells vergleichsweise niedrig. Um ein Vielfaches höher liegt der ermittelte JavaScript-Wert (1.103,55), welcher jedoch abermals deutlich vom CSS-Wert übertroffen wird (1.928,89).

Die mit Abstand größte Streuung um den präsentierten Mittelwert lässt sich für HTML-Smells feststellen (CV = 211,21%). Für die anderen Kategorien liegt dieser Wert deutlich tiefer. Die Vorteile der großen Stichprobe werden bei der Betrachtung der ermittelten Konfidenzintervalle um den Mittelwert deutlich. Für alle Kategorien, mit Ausnahme von HTML, liegt der tatsächliche Mittelwert mit einer Wahrscheinlichkeit von 95% im Bereich

von $\pm \leq 1,275$ des errechneten Mittelwerts. Aus der höheren Standardabweichung resultiert für HTML auch ein größeres Konfidenzintervall. Mit einer Wahrscheinlichkeit von 95% liegt der tatsächliche Mittelwert hier in einem Bereich von 190,56 bis 206,80.

Die Mittelwerte beziehen noch nicht ein, dass unterschiedlich viele Smells in den einzelnen Kategorien untersucht wurden. Daher listet Tabelle 7.4 zusätzlich den $\frac{\tilde{x}}{NoAS}$-Wert. Er gibt Auskunft darüber, wie häufig ein einzelner durchschnittlicher Smell im Mittel auf einer Website auftritt.

In Anbetracht aller untersuchten Sprachen tritt ein durchschnittlicher Code Smell 74,17-mal je Website auf. Ein *SoC*-Smell ist im Mittel 13,30-mal auf einer Website vertreten. Ein HTML-Smell tritt mit 24,83 Instanzen je Website fast doppelt so häufig auf. Ein JavaScript-Smell ist durchschnittlich sogar 64,91-mal auf einer einzelnen Website vertreten. Mit deutlichem Abstand tritt ein CSS-Smell am häufigsten auf. Dieser ließ sich im Mittel 160,74-mal auf einer einzelnen Website identifizieren.

Teilergebnis RQ.II)

Wie häufig treten Code Smells im Web insgesamt auf? Gibt es deutliche Unterschiede zwischen den untersuchten Sprachen?

Die 39 untersuchten Code Smells traten durchschnittlich 3.337,53-mal je Website auf. Ein einzelner Code Smell ist im Durchschnitt 74,17-mal auf einer Website identifiziert worden.
Ein durchschnittlicher JavaScript-Smell tritt je Website mehr als doppelt so häufig wie ein HTML-Smell auf. Ein CSS-Smell ist im Mittel wiederum mehr als doppelt so häufig wie ein JavaScript-Smell identifiziert worden. Er ist damit mit Abstand am häufigsten vertreten. Es ließen sich somit deutliche Unterschiede in der Smell-Verbreitung zwischen den einzelnen Sprachen feststellen.
SoC-Smells treten weniger oft auf als Smells in den einzelnen Sprachen. Ihr Anteil an der Gesamtheit aller Smells ist jedoch noch immer als sehr relevant anzusehen.

Durch die Erhebung der analysierten Datei-, Zeilen- und Zeichenanzahlen je Website kann die durchschnittliche Verbreitung der Code Smells noch feiner als auf der Website-Ebene berechnet werden (⇒ Tabelle 7.5). Die Aufschlüsselung der verschiedenen Werte, die in Tabelle 7.4 dargestellt sind, ist auf diesen Stufen allerdings nicht möglich. Ursache hierfür ist, dass Code Smells selten in einzelnen Zeilen und nahezu nie in einzelnen Zeichen auftreten können, da ihr Code-Umfang größer ist. Somit ist auch eine Erhebung einzelner Smells je Zeile oder Zeichen nicht möglich. Werte, wie die Streuung um den Mittelwert, können somit für diese Ebenen nicht erhoben werden. Eine separate Erhebung für die einzelnen Dateien fand darüber hinaus nicht statt.

Aufgrund der besseren Anschaulichkeit werden die Durchschnittswerte je Datei anders ermittelt als jene für Zeilen und Zeichen. Der *Smells per File*-Wert (SpF) gibt an, wie viele Smells durchschnittlich in einer Datei identifiziert wurden. Entsprechend zeugt ein

niedrigerer Wert von einer höheren Code-Qualität. Die *Lines per Smell*- (LpS) und *Characters per Smell*-Werte (CpS) geben hingegen an, wie viele Zeilen oder Zeichen durchschnittlich für eine einzelne Smell-Instanz vorhanden sind. Somit ist hier ein höherer Wert Repräsentant einer ebenso höheren Code-Qualität. Außerdem müssen die Kennwerte für Zeilen und Zeichen für Erkenntnisse über einen „durchschnittlichen Smell" mit dem *NoAS*-Wert multipliziert werden, statt durch ihn dividiert zu werden.

Kategorie	SpF	$\frac{SpF}{NoAS}$	LpS	$LpS \cdot NoAS$	CpS	$CpS \cdot NoAS$
HTML	198,68	24,83	4,87	38,96	458,52	3.668,16
CSS	841,31	70,11	0,66	7,94	72,79	873,51
JavaScript	361,31	21,25	1,23	20,93	239,38	4.069,48
SoC*	70,31	8,79	10,01	80,10	1.262,63	10.101,04
Gesamt	525,84	11,69	1,08	48,57	148,52	6.683,20

Tabelle 7.5.: Durchschnittliche Smell-Verbreitung bezüglich Dateien, Zeilen und Zeichen

* Die SoC-Werte für die Anzahl an Dateien, Zeilen und Zeichen ergaben sich abermals rechnerisch
SpF: Smells per File, LpS: Lines per Smell, CpS: Characters per Smell, NoAS: Number of Analyzed Smells

Je untersuchter Datei wurden durchschnittlich 525,84 Smells identifiziert. Bezüglich der Kategorien zeigt sich die gleiche Reihenfolge wie bei den Websites. So wurden mit 70,31 *SoC*-Smells die wenigsten je Datei berechnet. Mit 198,68 wurden deutlich mehr HTML-Smells je Datei identifiziert. Übertroffen wurde dieser Wert von den JavaScript-Smells (361,31). Mit deutlichem Abstand am höchsten ist der CSS-Wert (841,31).

Ein durchschnittlicher Code Smell tritt somit je durchschnittlicher Web-Datei 11,69-mal auf. In diesem Kontext zeigt sich in der bisher einheitlich ermittelten Kategorie-Reihenfolge erstmals ein Unterschied bezüglich der Smell-Häufigkeit. Da mehr JavaScript-Smells als HTML-Smells untersucht wurden, ist die Anzahl eines JavaScript-Smells je Datei (21,25) niedriger als die eines HTML-Smells (24,83), obwohl die Anzahl aller identifizierten Smell-Instanzen je Datei in JavaScript höher liegt.

Der Gesamt-*LpS*-Wert veranschaulicht erstmals sehr deutlich, welch hohe Verbreitung von Smells in dieser Analyse nachgewiesen wurde. Durchschnittlich wurde alle 1,08 Zeilen ein Smell identifiziert – also umgerechnet nahezu ein Smell in jeder analysierten Zeile. Während ein *SoC*-Smell nur alle 10,01 und ein HTML-Smell alle 4,87 Zeilen auftritt, liegt der Wert für JavaScript bereits relativ nahe an einem Smell je Zeile (1,23). Bezüglich CSS-Smells tritt ein Smell sogar alle 0,66 Zeilen auf. Insgesamt wurden somit deutlich mehr CSS-Smells identifiziert, als entsprechende Code-Zeilen analysiert wurden.

Insgesamt tritt ein einzelner Smell damit durchschnittlich ein Mal in einem 48,57 Zeilen langem Code-Block auf. Ein *SoC*-Smell ist dabei abermals unterdurchschnittlich oft vertreten (alle 80,10 Zeilen). In den Kategorien HTML (alle 38,96 Zeilen), JavaScript (alle 20,93 Zeilen) und CSS (alle 7,94 Zeilen) tritt ein einzelner Smell überdurchschnittlich oft

auf. Auch hier hebt sich der ermittelte CSS-Wert wieder deutlich ab.

Wie in Kapitel 7.1 bereits beschrieben, handelt es sich bei den *CpS*-Werten wohl um die aussagekräftigsten Ergebnisse. Insgesamt wurde alle 148,52 Zeichen ein Smell identifiziert. *SoC*-Smells treten auch in dieser Betrachtung am seltensten auf (alle 1.262,63 Zeichen). Die untersuchten HTML-Smells wurden von den drei Sprachen auch bezüglich der Zeichen-Anzahl am wenigsten häufig entdeckt (alle 458,52 Zeichen). Einer der JavaScript-Smells war ein mal in durchschnittlich 239,38 Zeichen vertreten. Deutlich am häufigsten wurden wiederum die CSS-Smells identifiziert (alle 72,79 Zeichen).

Multipliziert mit dem *NoAS*-Wert zeigt sich, dass ein durchschnittlicher Code Smell im Web somit ein mal in 6.683,20 Zeichen aufzufinden ist. Abermals schneidet nur der durchschnittliche *SoC*-Smell besser als die Gesamtheit ab – er tritt alle 10.101,04 Zeichen auf. Ein entscheidender Unterschied zu anderen Kennzahlen zeigt sich bei dieser besonders aussagekräftigen Metrik bei dem Vergleich der Smell-Verbreitung zwischen JavaScript und HTML: Mit einer Instanz in allen 3.668,16 Zeichen tritt der durchschnittliche HTML-Smell häufiger auf als ein solcher JavaScript-Smell (alle 4.069,48 Zeichen). Auch bei dieser Kennzahl wird deutlich, dass CSS-Smells viel stärker verbreitet sind als Smells in den anderen Kategorien. Ein CSS-Smell kann bereits in einem Code-Block, der lediglich 873,51 Zeichen umfasst, durchschnittlich ein Mal gefunden werden. Somit sind CSS-Smells etwa 7,65-mal häufiger vertreten als ein durchschnittlicher Smell in der Gesamtheit der Web-Sprachen und etwa 4,66-mal häufiger als JavaScript-Smells – auf denen dennoch der Fokus anderer Tools und der Forschung liegt.

Teilergebnis RQ.II)

Wie häufig treten Code Smells im Web insgesamt auf? Gibt es deutliche Unterschiede zwischen den untersuchten Sprachen?

Eine durchschnittliche Web-Datei enthält 525,84 Instanzen der untersuchten Code Smells. Im Mittel tritt eine Instanz der 39 untersuchten Smells in nahezu jeder Zeile und in allen 148,52 Zeichen Code auf.
Betrachtet man einen einzelnen Smell, ist dieser durchschnittlich 11,69-mal je Datei vertreten. Er ist im Mittel ein Mal in 48,57 Zeilen oder 6.683,20 Zeichen vorhanden. Am häufigsten tritt ein einzelner durchschnittlicher Smells mit Abstand in CSS auf (alle 873,51 Zeichen). Auch in HTML (alle 3.668,16 Zeichen) ist ein einzelner Smell im Mittel häufiger identifiziert worden als in der Programmiersprache JavaScript (alle 4.069,48 Zeichen). Ein *SoC*-Smell wurde durchschnittlich weniger oft identifiziert als die Smells in den untersuchten Sprachen (alle 10.101,04 Zeichen).
In Bezug auf Smells je Datei zeigt sich die gleiche Reihenfolge dieser Kategorien. Lediglich bei der Anwendung von Code-Zeilen als Maß gibt es eine Verschiebung. Hier ist ein JS-Smell im Mittel häufiger als ein HTML-Smell, da Zeilen in JavaScript durchschnittlich deutlich länger sind.

7.3. Verbreitung der einzelnen Code Smells im Web

7.3.1. HTML-Smells

Betrachtet man die individuelle Verbreitung der Smells in einer einzelnen Sprache, gewinnt der relative Anteil an der Gesamtheit deutlich an Bedeutung. Da den Unterschieden der Code-Metriken in den verschiedenen Sprachen (⇒ Tabelle 7.2) bei diesem Vergleich keine Bedeutung mehr zukommt, repräsentieren die *Smells per Website-*, *Smells per File-*, *Lines per Smell-* und *Characters per Smell*-Werte lediglich das gleiche Verhältnis, welches auch aus den absoluten und relativen Werten abzulesen ist. Die beschriebenen Werte tragen bei dieser Art des Vergleiches somit nur noch zur Veranschaulichung der Verbreitung auf Code-Ebene bei. In der folgenden schriftlichen Präsentation der Ergebnisse kommt ihnen daher eine eher untergeordnete Rolle zu.

Bezüglich der Verbreitung der untersuchten HTML-Smells (⇒ Tabelle 7.6) sticht der Smell *Lengthy Lines* extrem hervor. Er alleine macht 89,59% aller gefundenen Instanzen von HTML-Smells aus und tritt im Mittel 177,99-mal auf einer einzelnen Website auf. Aus diesem extremen Ergebnis lässt sich unmittelbar die Notwendigkeit des verstärkten Einsatzes von Entwicklungsumgebungen ableiten, die HTML-Zeilen sinnvoll aufteilen.

	Gesamt		Je Website					Weitere Mittelwerte		
Smell	ISI	%	MAX	$\bar{x}$	CV	95%-Konf.-int. ⇓	95%-Konf.-int. ⇑	SpF	LpS	CpS
Lengthy Lines	1.8253M	89,59	16.153	177,99	212,53%	170,67	185,31	177,99	5,44	511,81
Opening Tag Missing Closing Tag	97.105	4,77	6.001	9,47	1.283,84%	7,12	11,82	9,47	102,19	9.620,62
Deprecated Attributes	57.671	2,83	3.405	5,62	975,36%	4,56	6,69	5,62	172,07	16.198,96
Skipped Headings	26.313	1,29	795	2,57	400,89%	2,37	2,76	2,57	377,14	35.503,76
Commented-Out Code	20.966	1,03	449	2,04	379,80%	1,89	2,19	2,04	473,32	44.558,35
Uppercase Tags	7.003	0,34	1.047	0,68	2.345,62%	0,37	0,99	0,68	1.417,05	133.401,45
Mysterious Name	2.761	0,14	43	0,27	479,36%	0,24	0,29	0,27	3.594,20	338.359,41
Deprecated Tags	315	0,02	143	0,03	4.809,93%	0,00	0,06	0,03	31.503,41	2,9657M

Tabelle 7.6.: Verbreitung der untersuchten HTML-Smells

ISI: Identified Smell Instances, %: Anteil an der Gesamtheit in Prozent, MAX: Maximalwert, $\bar{x}$: Mittelwert, CV: Variationskoeffizient, Konf.-int.: Konfidenzintervall, ⇓: Untergrenze, ⇑: Obergrenze, SpF: Smells per File, LpS: Lines per Smell, CpS: Characters per Smell, M: Millionen
Der Minimalwert liegt jeweils bei 0, weshalb auf eine explizite Darstellung in der Tabelle verzichtet wurde.

Mit weitem Abstand folgen die Smells *Opening Tag Missing Closing Tag* (4,77%), *Deprecated Attributes* (2,83%), *Skipped Headings* (1,29%) und *Commented-Out Code* (1,03%). Der relative Anteil von 1% – 5% resultiert in einem Mittelwert von 2 – 10 Instanzen je Website.

Am seltensten sind die Smells *Uppercase Tags* (0,34%), *Mysterious Name* (0,14%) und *Deprecated Tags* (0,02%) identifiziert worden. Sie treten – teilweise deutlich – seltener als ein Mal je Website auf.

Bezüglich HTML-Smells gleichen sich die *Smells per Website*- und *Smells per File*-Werte, da eine Web Page aus je einer HTML-Datei besteht. Der extreme Unterschied in der Verbreitung der einzelnen HTML-Smells zeigt sich nochmals in den ermittelten *LpS*- und *CpS*-Werten. Während der Smell *Lenghty Lines* durchschnittlich alle 5,44 Zeilen (entspricht 511,81 Zeichen) auftritt, ist der zweithäufigste Smell *Opening Tag Missing Closing Tag* 19-mal seltener identifiziert worden (alle 102,19 Zeilen, beziehungsweise alle 9.620,62 Zeichen). Bei dem am wenigsten häufig identifizierten Smell *Deprecated Tags* übersteigt der Kennwert für Zeilen den Wert 30.000 (31.503,41). Er tritt somit lediglich ein mal in fast 3 Mio. Zeichen an Code auf (2,9675M).

Durch das Speichern der exakten Smell-Instanzen für jede analysierte Website lässt sich neben den angegebenen Konfidenzintervallen auch der Variationskoeffizient (CV) berechnen. Er beschreibt das relative Verhältnis der Streuung aller Werte um den angegebenen Mittelwert und ermöglicht somit einen fairen Vergleich der Streuung zwischen den verschiedenen Smells.

Es zeigt sich, dass der Smell *Lengthy Lines* neben der stärksten Verbreitung auch die geringste Streuung aufweist (212,53%). Der Smell ist somit zusätzlich von allen untersuchten HTML-Smells am konstantesten über alle Websites hinweg mit einem relativ ähnlichen Anteil vertreten.

Die extrem hohen Werte der Smells *Uppercase Tags* und *Deprecated Tags* zeugen von einer Verbreitung der Art *„Ganz oder gar nicht“*. Entweder sind sich Entwickler*innen dieser Code Smells bewusst und verhindern ihre Entstehung vollständig oder aber nicht und sie werden sehr häufig eingebaut. Ein Blick in die weiteren Daten außerhalb von Tabelle 7.6, bestätigt diesen Eindruck. Bei 97,17% der Websites wurde der Smell *Uppercase Tags* exakt 0-mal nachgewiesen. Die Nutzung veralteter Tags konnte sogar auf 99,48% aller Websites nicht festgestellt werden. Für die Smells *Opening Tag Missing Closing Tag* und *Deprecated Attributes* wurde ebenfalls ein hoher, jedoch weniger extremer Variationskoeffizient berechnet. Im Fall der Smells *Deprecated Tags* und *Deprecated Attributes* könnte die hohe Variation zusätzlich damit zusammenhängen, dass von dem Crawler ein paar Websites aufgespürt worden sein könnten, welche zu Zeiten entstanden, als die Tags und Attribute noch nicht als veraltet galten und welche seither nicht überarbeitet wurden. Bezüglich *Uppercase Tags* ist die aufgezeigte *„Ganz oder gar nicht“*-Verteilung ebenfalls logisch nachvollziehbar. Entweder schreibt man alle HTML-Tags durchgehend groß oder durchgehend klein – ein Mix aus beiden Versionen dürfte eher untypisch sein.

Teilergebnis RQ.III)

Wie stark sind die einzelnen Code Smells verbreitet?

Bezüglich der untersuchten HTML-Smells sticht *Lenghty Lines* extrem heraus – er macht 90% der identifizierten Smell-Instanzen aus. Bei mehr als 1% Anteil liegen *Opening Tag Missing Closing Tag*, *Deprecated Attributes*, *Skipped Headings* und *Commented-Out Code*. Den drei weiteren untersuchten Smells konnte mit einem relativen Anteil von unter 0,35% keine auffallend problematische Verbreitung nachgewiesen werden.

7.3.2. CSS-Smells

In der Kategorie der CSS-Smells lassen sich mit Abstand die höchsten Werte für gefundene Smell-Instanzen feststellen. Mit *Duplicated Code* (6,0440M), *Properties with Hard-Coded Values* (5,9735M) und *High Specificity Values* (3,9165M) übersteigen drei CSS-Smells die Höchstwerte aller anderen Kategorien deutlich.

Dennoch sind die CSS-Smells deutlich ausgeglichener verteilt als es in der Kategorie HTML der Fall war. Die Smells *Duplicated Code* (30,55%) und *Properties with Hard-Coded Values* (30,20%) führen die Liste mit ähnlichem Anteil an. Je Website sind sie im Mittel über 580-mal vertreten – für einzelne Smells ist dies ein extrem hoher Wert. Fast jede zweite Code-Zeile in CSS enthält duplizierten Code (2,17) oder nutzt absolute Einheiten (2,19). Es folgt der Smell *High Specificity Values*, welcher mit 19,80% zur Gesamtheit der CSS-Smells beiträgt. Dies entspricht einer Verbreitung von durchschnittlich 381,91 Instanzen je Website und einer Instanz in 3,34 Zeilen Code. Für die drei beschriebenen dominanten Smells ergibt sich zusätzlich ein vergleichsweise geringer Variationskoeffizient. Demnach wurde die hohe Verbreitung auch sehr konstant über die verschiedenen Websites hinweg nachgewiesen.

Zwischen 1% und 10% der Gesamtheit aller CSS-Smells machen *Usage of !important* (7,78%), *Too Long Rules* (5,22%), *Too Much Cascading* (3,22%) und *Lengthy Lines* (1,63%) aus. Aufgrund der insgesamt hohen Verbreitung von CSS-Smells resultieren diese eher geringen relativen Anteile dennoch in hohen Werten von durchschnittlich 30 – 150 Instanzen je Website.

Mit einem relativen Anteil von weniger als 1% scheinen *Universal Selectors* (0,61%), *Selectors with ID and Class or Element* (0,47%), *Empty Rules* (0,21%), *Commented-Out Code* (0,18%) und *Too General Selectors* (0,13%), in Hinblick auf ihre Verbreitung, die am wenigsten problematischen CSS-Smells zu sein. Die genannten Anteile resultieren im Mittel in 2,43 – 11,81 Instanzen je Website, womit dennoch die Werte der meisten HTML-Smells überstiegen werden. Dieser Vergleich zeigt nochmals, dass über die Gesamtheit aller CSS-Smells hinweg eine extrem hohe Verbreitung – und damit eine sehr schlechte Code-Qualität – nachgewiesen wurde.

	Gesamt		Je Website					Weitere Mittelwerte		
Smell	ISI	%	MAX	$\bar{x}$	CV	95%-Konf.-int. ⇓	95%-Konf.-int. ⇑	SpF	LpS	CpS
Duplicated Code	6,0440M	30,55	21.944	589,37	221,05%	564,15	614,58	257,06	2,17	238,24
Properties w. Hard-Coded Values	5,9735M	30,20	16.333	582,49	192,22%	560,82	604,16	254,06	2,19	241,05
High Specificity Values	3,9165M	19,80	14.979	381,91	231,53%	364,80	399,03	166,58	3,34	367,65
Usage of !important	1,5389M	7,78	18.459	150,06	302,06%	141,29	158,84	65,45	8,50	935,66
Too Long Rules	1,0334M	5,22	1.747	100,77	184,37%	97,17	104,36	43,95	12,66	1.393,39
Too Much Cascading	637.385	3,22	5.989	62,15	392,47%	57,43	66,87	27,11	20,53	2.259,07
Lengthy Lines	321.633	1,63	6.193	31,36	505,13%	28,30	34,43	13,68	40,68	4.476,83
Universal Selectors	121.091	0,61	1.773	11,81	331,03%	11,05	12,56	5,15	108,06	11.891,03
Selectors with ID and Class or Element	92.561	0,47	1.120	9,03	297,77%	8,51	9,55	3,94	141,37	15.556,19
Empty Rules	41.245	0,21	437	4,02	531,26%	3,61	4,44	1,75	317,26	34.910,82
Commented-Out Code	35.660	0,18	858	3,48	528,46%	3,12	3,83	1,52	366,95	40.378,49
Too General Selectors	24.962	0,13	52	2,43	120.87%	2,38	2,49	1,06	524,22	57.683,56

Tabelle 7.7.: Verbreitung der untersuchten CSS-Smells

ISI: Identified Smell Instances, %: Anteil an der Gesamtheit in Prozent, MAX: Maximalwert, $\bar{x}$: Mittelwert, CV: Variationskoeffizient, Konf.-int.: Konfidenzintervall, ⇓: Untergrenze, ⇑: Obergrenze, SpF: Smells per File, LpS: Lines per Smell, CpS: Characters per Smell, M: Millionen
Der Minimalwert liegt jeweils bei 0, weshalb auf eine explizite Darstellung in der Tabelle verzichtet wurde.

Bezüglich des Variationskoeffizienten übersteigen drei Smells einen Wert von 500%: *Lenghty Lines*, *Empty Rules* und *Commented-Out Code*. Es fällt auf, dass alle drei Smells mit dem „*Sauberhalten*“ des Codes durch sinnvolle Formatierung und die Beseitigung unnötigen Codes zu tun haben. Um diesem ersten Eindruck weiter nachgehen zu können, wurde entschieden, dass die Code Smells hinsichtlich ihrer zugeordneten Labels nicht nur bezüglich ihrer Verbreitung, sondern auch hinsichtlich ihrer Streuung analysiert werden sollen (⇒ Kapitel 7.4).

Teilergebnis RQ.III)

Wie stark sind die einzelnen Code Smells verbreitet?

Am häufigsten wurden die CSS-Smells *Duplicated Code*, *Properties with Hard-Coded Values* und *High Specificity Values* nachgewiesen. Aufgrund der extrem hohen Gesamtheit an identifizierten Smell-Instanzen, resultieren auch die geringeren relativen Anteile aller anderen untersuchten CSS-Smells in hohen Verbreitungswerten. Keiner der untersuchten Smells tritt im Mittel weniger als 2-mal je Website auf. Fünf der zwölf Smells sind im Durchschnitt sogar mehr als 100-mal je Website zu finden.

7.3.3. JavaScript-Smells

Die am häufigsten identifizierten JavaScript-Smells sind *Excessive Global Variables* (24,92%, 274,96-mal je Website) und *Depth* (24,33%, 268,44-mal je Website). Sie treten im Mittel ungefähr in einem Code-Block von fünf Zeilen und weniger als 1.000 Zeichen ein Mal auf. Die Anzahl identifizierter Instanzen (ISI) liegt bei beiden Smells jedoch noch deutlich tiefer als bei den häufigsten CSS-Smells.

Einen relativen Anteil zwischen 10% und 20% weisen die Smells *Argument Count Mismatch* (17,02%), *Lenghty Lines* (15,45%) und *Mysterious Name* (10,06%) auf. Dies resultiert dennoch in sehr hohen Werten im Bereich von etwa 110 bis 188 Instanzen je Website.

Im Vergleich zu den anderen Werten, ist der Smell *This Assign* relativ alleinstehend. Sein Wert von 3% ist der einzige, der unter 10%, aber deutlich über 1% liegt.

Es folgt eine ganze Reihe von elf Smells (von insgesamt 17), deren Anteile nahe oder unter der 1%-Marke liegt. Die daraus folgende Verbreitung von durchschnittlich 1,76 – 12,04 Instanzen je Website ist jedoch nach wie vor als relevant anzusehen und liegt deutlich über einigen Werten der analysierten HTML-Smells.

Für JavaScript ergibt sich somit ein interessantes Gesamtbild. Während keinem einzigen der 17 Smells eine auffallend geringe Verbreitung nachgewiesen wurde, weist ein breiter Block von elf Smells eine relativ niedrige Verbreitung auf. Fünf weitere Smells weisen eine starke Verbreitung, mit einem relativen Anteil von 10% bis 25% auf – auch hier gibt es aber keinen extremen Ausreißer nach oben, wie es etwa bei den HTML-Smells der Fall war. Zwischen diesen Blöcken von schwach und stark verbreitenden Smells positioniert sich mit *Empty Catch* nur ein alleinstehender Smell.

Hinsichtlich der Wertstreuung ergaben sich für die stärker verbreiteten Smells relativ niedrige Variationskoeffizienten. Somit zeigt sich hier eine recht konstante Verbreitung, auf hohem Niveau, über alle untersuchten Websites hinweg. Auffallend hohe Werte weisen die Smells *Commented-Out Code* (445,77%), *Duplicated Code* (550,73%) und *Repeated Switches* (427,94%) auf. Auch hier lässt sich mit dem Zusammenhang zur Code-Wiederholung eine auffällige Gemeinsamkeit ausmachen, welche für nähere Untersuchungen spricht (⇒ Kapitel 7.4).

	Gesamt		Je Website					Weitere Mittelwerte		
Smell	ISI	%	MAX	$\bar{x}$	CV	95%-Konf.-int. ⇓	95%-Konf.-int. ⇑	SpF	LpS	CpS
Excessive Global Variables	2,8197M	24,92	3.232	274,96	130.26%	268,03	281,89	90,02	4,94	960,75
Depth	2,7529M	24,33	8.430	268,44	190,42%	258,55	278,34	87,89	5,06	984,08
Argument Count Mismatch	1,9265M	17,02	9.916	187,86	195,08%	180,76	194,95	61,51	7,23	1.406,22
Lengthy Lines	1,7485M	15,45	4.705	170,50	173,37%	164,78	176,22	55,82	7,97	1.549,34
Mysterious Name	1,1381M	10,06	5.603	110,98	222,83%	106,20	115,77	36,34	12,24	2.380,22
This Assign	339.539	3,00	1.723	33,11	200,22%	31,83	34,39	10,84	41,04	7.978,59
Empty Catch	123.436	1,09	373	12,04	155,82%	11,67	12,40	3,94	112,89	21.946,94
Commented-Out Code	97.998	0,87	2.391	9,56	445,77%	8,73	10,38	3,13	142,19	27.643,85
Assignment in Conditional Statement	77.438	0,68	353	7,55	189,88%	7,27	7,83	2,47	179,95	34.983,37
Duplicated Code	72.043	0,64	2.283	7,03	550,73%	6,28	7,77	2,30	193,42	37.603,13
Long Parameter List	61.321	0,54	375	5,98	227.46%	5,72	6,24	1,96	227,24	44.178,05
Complex Switch Case	35.924	0,32	232	3,50	251.35%	3,33	3,67	1,15	387,89	75.410,37
Repeated Switches	29.697	0,26	374	2,90	427,94%	2,66	3,14	0,95	469,23	91.222,75
Long Method	28.028	0,25	170	2,73	346,80%	2,55	2,92	0,89	497,17	96.654,85
Nested Callbacks	24.784	0,22	431	2,42	353,39%	2,25	2,58	0,79	562,24	109.306,09
Array Length Assignment	22.904	0,20	180	2,23	253,33%	2,12	2,34	0,73	608,39	118.278,12
Chained Methods	18.015	0,16	296	1,76	299,11%	1,66	1,86	0,58	773,50	150.377,03

Tabelle 7.8.: Verbreitung der untersuchten JavaScript-Smells

ISI: Identified Smell Instances, %: Anteil an der Gesamtheit in Prozent, MAX: Maximalwert, $\bar{x}$: Mittelwert, CV: Variationskoeffizient, Konf.-int.: Konfidenzintervall, ⇓: Untergrenze, ⇑: Obergrenze, SpF: Smells per File, LpS: Lines per Smell, CpS: Characters per Smell, M: Millionen
Der Minimalwert liegt jeweils bei 0, weshalb auf eine explizite Darstellung in der Tabelle verzichtet wurde.

Teilergebnis RQ.III)

Wie stark sind die einzelnen Code Smells verbreitet?

Die häufigsten JavaScript-Smells sind *Excessive Global Variables*, *Depth*, *Argument Count Mismatch*, *Lenghty Lines* und *Mysterious Name*. Nach einer größeren und schwach vertretenden Lücke folgt eine Vielzahl von Smells im Bereich von um oder unter 1% relativen Anteils. Weder im obersten noch im untersten Bereich gibt es einen extremen Ausreißer.

7.3.4. *Separation of Concern*-Smells

Da sechs der analysierten *SoC*-Smells in HTML und zwei in JavaScript identifiziert werden können, gewinnt der *CpS*-Wert als Vergleichsmaß wieder an Bedeutung und liefert aussagekräftigere Ergebnisse als der relative Anteil an der Gesamtheit.

Die Unterschiede zwischen den beiden Werten werden schnell deutlich: Während *HTML in JavaScript* vom relativen Anteil her der am stärksten vertretene *SoC*-Smell ist (35,90%, alle 6.914,21 Zeichen), tritt *CSS in HTML Style Attribute* gemessen an der Zeichenanzahl mehr als doppelt so häufig auf (26,64%, jedoch bereits alle 3.213,26 Zeichen).

	Gesamt		**Je Website**					**Weitere Mittelwerte**		
Smell	**ISI**	**%**	MAX	$\bar{x}$	CV	**95%-Konf.-int.** ⇓	⇑	**SpF**	**LpS**	**CpS**
CSS in HTML Style Attribute	290.736	26,64	6.247	28,35	383,24%	26,25	30,45	28,35	34,13	3.213,26
HTML in JavaScript	391.808	35,90	3.687	38,21	241,36%	36,42	39,99	12,51	35,57	6.914,21
JavaScript in HTML Script Tag	91.410	8,38	221	8,91	129,13%	8,69	9,14	8,91	108,56	10.220,00
CSS in JavaScript	220.726	20,23	2.424	21,52	222,92%	20,60	22,45	7,05	63,13	12.273,33
CSS in HTML Style Tag	46.234	4,24	404	4,51	316,63%	4,23	4,78	4,51	214,64	20.206,13
JavaScript in HTML Events	35.533	3,26	3.423	3,46	1.080,39%	2,74	4,19	3,46	279,28	26.291,34
Deprecated Styling Tags	7.660	0,70	1.467	0,75	2.567,19%	0,38	1,12	0,75	1.295,51	121.959,57
JavaScript in HTML Links	7.201	0,66	184	0,70	734,72%	0,60	0,80	0,70	1.378,08	129.733,41

Tabelle 7.9.: Verbreitung der untersuchten SoC-Smells

ISI: Identified Smell Instances, %: Anteil an der Gesamtheit in Prozent, MAX: Maximalwert, $\bar{x}$: Mittelwert, CV: Variationskoeffizient, Konf.-int.: Konfidenzintervall, ⇓: Untergrenze, ⇑: Obergrenze, SpF: Smells per File, LpS: Lines per Smell, CpS: Characters per Smell
Der Minimalwert liegt jeweils bei 0, weshalb auf eine explizite Darstellung in der Tabelle verzichtet wurde.

Es folgen mit relativ ähnlichen *CpS*-Werten die Smells *JavaScript in HTML Script Tag* (alle 10.220,00 Zeichen) und *CSS in JavaScript* (alle 12.273,33 Zeichen). Während sich *CSS in HTML Style Tag* (alle 20.206,13 Zeichen) und *JavaScript in HTML Events* (alle 26.291,34 Zeichen) im unteren Mittelfeld positionieren, sind die *SoC*-Smells *Deprecated Styling Tags* (alle 121.959,57 Zeichen) und *JavaScript in HTML Links* (alle 129.733,41 Zeichen) mit ähnlicher Verbreitung weit abgeschlagen.

Für die drei am seltensten identifizierten SoC-Smells lassen sich sehr hohe Variationskoeffizienten feststellen. Ein extremer Wert (2.567,19%) steht anhand des Smells *Deprecated Styling Tags* abermals mit veralteten Coding-Praktiken im Zusammenhang. Dieser Smell konnte auf 94,40% aller untersuchten Websites überhaupt nicht identifiziert werden. Diese Erkenntnisse decken sich sehr mit denen in der Kategorie der HTML-Smells (⇒ Kapitel 7.3.1).

Anhand von Tabelle 7.9 lässt sich außerdem feststellen, dass sich, hinsichtlich der Verbreitung einzelner Smells, keine Durchmischung von zwei der drei untersuchten Sprachen besonders abhebt. Es fällt lediglich auf, dass beide Smells, die in JavaScript auftreten, relativ hoch platziert sind. Addiert man die *ISI*-Werte der einzelnen Smells jedoch – je nachdem, welche zwei Sprachen sie betreffen – zeigen sich einige Unterschiede, welche in Abbildung 7.1 dargestellt werden.

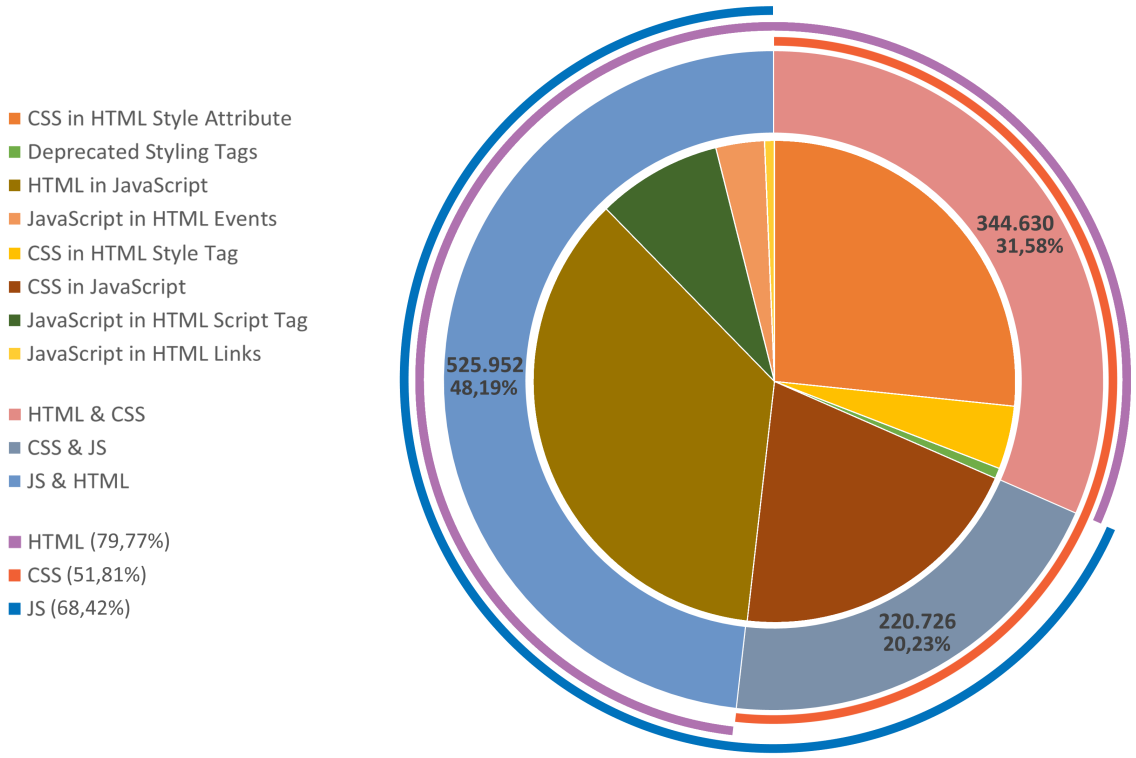


Abbildung 7.1.: Aggregierte Verbreitung der SoC-Smells nach vermischten Sprachen

Die äußeren drei dünnen Ringe der Grafik zeigen, an welchem Anteil der Gesamtheit aller *SoC*-Smells die jeweiligen drei Sprachen beteiligt sind. Mit 79,77% wird HTML

am häufigsten mit einer der beiden anderen Sprachen durchmischt. Am zweithäufigsten wird JavaScript unsauber von den anderen Sprachen separiert (68,42%). Das Schlusslicht bildet CSS mit einer Beteiligung an 51,81% aller *SoC*-Smells. Zu beachten ist hierbei, dass die Summe aller relativen Anteile bei 200% statt 100% liegt, da jeweils zwei Sprachen je Smell durchmischt werden. Während sich die geschilderte Reihenfolge klar ergeben hat, sind die Unterschiede dennoch klein genug, dass man von einem relativ ausgeglichenen Verhältnis sprechen kann, bei dem zumindest keine der Sprachen einen extremen Ausreißer darstellt.

Deutlicher werden die Unterschiede, wenn man nach den jeweiligen Kombinationen von zwei der drei Sprachen schaut – in Abbildung 7.1 durch den breiten Ring in der Mitte visualisiert. Es zeigt sich, dass fast jeder zweite Verstoß (48,19%) gegen das *Separation of Concerns*-Prinzip auf das Konto einer Durchmischung von HTML und JavaScript geht. Mit einigem Abstand folgen Smells, welche die unangebrachte Kombination von HTML und CSS thematisieren. Ihr Anteil an der Gesamtheit aller *SoC*-Smells liegt bei 31,58%. Am wenigsten häufig wurde die Kombination von CSS und JavaScript identifiziert. Gemessen daran, dass es auch nur eine Möglichkeit der Durchmischung dieser Sprachen gibt (⇒ beschrieben anhand des Smells *CSS in JavaScript*), ist der Anteil von 20,23% an der Gesamtheit aller *SoC*-Smells dennoch recht hoch.

Teilergebnis RQ.III)

Wie stark sind die einzelnen Code Smells verbreitet?

Die häufigsten Smells, welche sich durch die unangebrachte Kombination von HTML, CSS und JavaScript ergeben, sind *CSS in HTML Style Attribute* und *HTML in JavaScript*.
Die meisten *SoC*-Smells betreffen die Sprache HTML, gefolgt von JavaScript und CSS. Die Kombination aus *HTML & JavaScript* ist dabei am häufigsten vertreten, gefolgt von *HTML & CSS*. Die Kombination aus *CSS & JavaScript* bildet bei dieser Analyse das Schlusslicht. Dennoch lässt sich eine relativ ausgeglichene Verteilung zwischen den Sprachen feststellen, ohne extreme Ausreißer nach oben oder unten.

7.3.5. Sprachübergreifende Verbreitung aller untersuchten Smells

Mit *Commented-Out Code*, *Duplicated Code*, *Lengthy Lines* und *Mysterious Name* befinden sich in der Auswahl vier Code Smells, welche in mehreren Sprachen identifiziert wurden. Aufgrund der unterschiedlichen Code-Formatierungen eignet sich der *CpS*-Wert erneut am besten, um die Verbreitung dieser Smells in den Sprachen miteinander zu vergleichen (⇒ Tabelle 7.10).

Hinsichtlich des Smells *Commented-Out Code* zeigt sich eine relativ gleichmäßige Verteilung über alle drei Sprachen hinweg. Am häufigsten wurde der Smell in JavaScript identifiziert (alle 22.643,85 Zeichen) – am seltensten in HTML (alle 44.558,35 Zeichen).

Smell	Sprache	ISI	SpW	SpF	LpS	CpS
Commented-Out Code	HTML	20.966	2,04	2,04	473,32	44.558,35
	CSS	35.660	3,48	1,52	366,95	40.378,49
	JavaScript	97.998	9,56	3,13	142,19	27.643,85
Duplicated Code	CSS	6.043.960	589,37	257,06	2,17	238,24
	JavaScript	72.043	7,03	2,30	193,42	37.603,13
Lengthy Lines	HTML	1.825.311	177,99	177,99	5,44	511,81
	CSS	321.633	31,36	13,68	40,68	4.476,83
	JavaScript	1.748.510	170,50	55,82	7,97	1.549,34
Mysterious Name	HTML	2.761	0,27	0,27	3.594,20	338.359,41
	JavaScript	1.138.148	110,98	36,34	12,24	2.380,22

Tabelle 7.10.: Vergleich der Verbreitung mehrsprachiger Smells

ISI: Identified Smell Instances, SpW: Smells per Website, SpF: Smells per File, LpS: Lines per Smell, CpS: Characters per Smell

Der Unterschied zwischen diesem Hoch- und Tiefpunkt liegt bei einem Faktor von 1,61 und damit deutlich geringer als bei den anderen Smells. In CSS tritt *Commented-Out Code* im Mittel alle 40.378,49 Zeichen auf.

Der Smell *Duplicated Code* lässt sich mit CSS und JavaScript nur auf zwei Sprachen sinnvoll anwenden. Dabei zeigen sich extreme Unterschiede. Während der Smell in JavaScript durchschnittlich alle 37.603,13 Zeichen auftaucht, wurde er in CSS im Mittel bereits in 238,24 Zeichen ein Mal identifiziert – und ist damit der am häufigsten identifizierte Smell in einer einzelnen Sprache. Dieser Unterschied äußert sich in einem Faktor von 157,84. Beachtlich ist auch, dass der Smell in CSS somit alle 2,17 Zeilen auftritt, obwohl einer der beiden untersuchten Duplikationsgegenstände (identische Deklarationsabschnitte) im Mittel wahrscheinlich deutlich mehr als zwei Zeilen umfassen dürfte. Diese Vermutung geht aus den analysierten Daten jedoch nicht hervor, sondern beruht auf den persönlichen Kenntnissen der Sprache.

Lenghty Lines ist am häufigsten in HTML vertreten (alle 511,81 Zeichen), gefolgt von JavaScript (alle 1.549,34 Zeichen) und CSS (alle 4.476,83 Zeichen). Der Unterschiedsfaktor zwischen den beiden Extremwerten liegt damit bei 8,75 und damit weder so niedrig wie bei *Commented-Out Code* noch so hoch wie bei *Duplicated Code*.

Bei *Mysterious Name* zeigen sich wieder sehr deutliche Unterschiede, welche vor allem in dem sehr niedrigen Wert für HTML begründet sind (alle 338.359,41 Zeichen). In JavaScript tritt der Smell 142,15-mal häufiger auf (alle 2.380,22 Zeichen).

Bei dieser Betrachtung zeigt sich, dass sich, bei den Smells, die in mehreren Sprachen identifiziert werden können, keine Muster, hinsichtlich der Verbreitung in den jeweiligen Sprachen, erkennen lassen. Bei nur vier untersuchten Smells war jede der drei untersuchten Sprachen jeweils mindestens ein mal am häufigsten und ein mal am seltensten vertreten. Aussagen der Art *„Die gleichen Code Smells scheinen in JavaScript grundsätzlich häufiger*

aufzutreten als in HTML“ lassen sich aus diesem Ergebnis somit nicht ableiten. Hingegen scheint es so, dass die Unterschiede zwischen den Sprachen sehr individuell von dem jeweiligen Code Smell abhängen. Gleichzeitig ist zu beachten, dass die Stichprobengröße mit vier Smells für derartige Aussagen auch sehr gering ist.

Für die einzelnen Kategorien wurde bereits betrachtet, welche Code Smells wie stark verbreitet sind. Doch welche Code Smells stellen nun insgesamt im Web – und somit in einer sprachübergreifenden Betrachtung – das größte Problem hinsichtlich ihrer Verbreitungswerte dar?

Durch die verschiedenen errechneten Mittelwerte wurde die Grundlage für einen fairen und differenzierten sprachübergreifenden Vergleich geschaffen. Am aussagekräftigsten ist hierbei wieder der errechnete *CpS*-Wert, da es sich um die kleinste gemessene Einheit handelt.

Der Vergleich der Verbreitung aller 39 analysierten Code Smells ist in Tabelle 7.11 dargestellt. Bei Smells, die in mehreren Sprachen identifiziert wurden, wurden die *ISI*- und *SpW*-Werte für die verschiedenen Sprachen aufsummiert. Zum faireren Vergleich wird auch jeweils ein Wert gelistet, bei dem diese Werte durch die Anzahl an Kategorien, in denen der Smell identifiziert wurde, dividiert wurden. Bei den *SpF*-, *LpS*- und *CpS*-Werten handelt es sich bereits um Werte, bei denen die Mehrsprachigkeit berücksichtigt ist. Für *SoC*-Smells wurde für die Berechnung dieser drei Mittelwerte der Code-Umfang jener Sprache genutzt, in welcher der Smell identifiziert wird (in Tabelle 7.11 durch ein I markiert).

Bezüglich der identifizierten Instanzen liegen 10 der 39 Code Smells bei einem Wert über 1.000.000: *Properties with Hard-Coded Values*, *High Specificity Values*, *Duplicated Code*, *Usage of !important*, *Excessive Global Variables*, *Depth*, *Lengthy Lines*, *Too Long Rules*, *Argument Count Mismatch* und *Mysterious Name*. Auch gemessen am *CpS*-Wert handelt es sich bei diesen zehn Smells um die häufigsten. Ihre Verbreitung resultiert in Mittelwerten von mehr als 100 Instanzen je Website und mehr als 30 Instanzen je Datei. Dies entspricht – mit Ausnahme des Smells *Mysterious Name* – einer Instanz in weniger als 13 Code-Zeilen und weniger als 1.500 Zeichen.

Innerhalb dieser negativ auszulegenden „Top Ten“ lassen sich ebenfalls einige signifikante Unterschiede im *CpS*-Wert feststellen. Bereits zwischen den drei oberen Smells gibt es recht deutliche Unterschiede. Am häufigsten von allen 39 Smells wurde *Properties with Hard-Coded Values* identifiziert (241,05). Trotz der extrem hohen Verbreitungswerte wurde der Smell noch 1,53-mal häufiger aufgefunden als der zweithäufigste Smell *High Specificity Values* (367,65). Mit noch deutlicherem Abstand folgt *Duplicated Code* (678,37).

Anschließend kann mit einem Abstand von mehreren hundert Zeichen eine Gruppe von gleich drei Smells mit *CpS*-Werten im Bereich zwischen 900 und 1.000 identifiziert werden: *Usage of !important*, *Excessive Global Variables* und *Depth*. Mit wiederum einem deutlichen Abstand folgen abermals drei nah beieinanderliegende Smells: *Lengthy Lines*

	Kategorien				Instanzen		Mittelwerte				
Smell	HTML	CSS	JS	SoC	ISI	$\frac{ISI}{Kat.}$	SpW	$\frac{SpW}{Kat.}$	SpF	LpS	CpS
Prop. w. Hard-Cod. Val.		✓			5,9735M	5,9735M	582,49	582,49	254,06	2,19	241,05
High Specificity Values		✓			3,9165M	3,9165M	381,91	381,91	166,58	3,34	367,65
Duplicated Code		✓	✓		6,1160M	3,0580M	596,39	298,20	111,54	4,42	678,37
Usage of !important		✓			1,5389M	1,5389M	150,06	150,06	65,45	8,50	935,66
Excessive Global Var.			✓		2,8197M	2,8197M	274,96	274,96	90,02	4,94	960,75
Depth			✓		2,7529M	2,7529M	268,44	268,44	87,89	5,06	984,08
Lengthy Lines	✓	✓	✓		3,8954M	1,2985M	379,86	126,62	59,85	9,48	1.304,89
Too Long Rules		✓			1,0334M	1,0334M	100,77	100,77	43,95	12,66	1.393,39
Arg. Count Mismatch			✓		1,9265M	1,9265M	187,86	187,86	61,51	7,23	1.406,22
Mysterious Name	✓	✓			1,1409M	570.455	111,25	55,63	33,79	20,17	2.080,89
Too Much Cascading		✓			637.385	637.385	62,15	62,15	27,11	20,53	2.259,07
CSS in HTML Style Attr.	I			✓	290.736	290.736	28,35	28,35	28,35	34,13	3.213,26
HTML in JavaScript			I	✓	391.808	391.808	38,21	38,21	12,51	35,57	6.914,21
This Assign			✓		339.539	339.539	33,11	33,11	10,84	41,04	7.978,59
Open. Tag Miss. Clos. T.	✓				97.105	97.105	9,47	9,47	9,47	102,19	9.620,62
JS in HTML Script Tag	I			✓	91.410	91.410	8,91	8,91	8,91	108,56	10.220,00
Universal Selectors		✓			121.091	121.091	11,81	11,81	5,15	108,06	11.891,03
CSS in JavaScript			I	✓	220.726	220.726	21,52	21,52	7,05	63,13	12.273,33
Sel. w. ID & Cls. or El.		✓			92.561	92.561	9,03	9,03	3,94	141,37	15.556,19
Deprecated Attributes	✓				57.671	57.671	5,62	5,62	5,62	172,07	16.198,96
CSS in HTML Style Tag	I			✓	46.234	46.234	4,51	4,51	4,51	214,64	20.206,13
Empty Catch			✓		123.436	123.436	12,04	12,04	3,94	112,89	21.946,94
JS in HTML Events	I			✓	35.533	35.533	3,46	3,46	3,46	279,28	26.291,34
Commented-Out Code	✓	✓	✓		154.624	51.541	15,08	5,03	2,38	238,93	32.874,26
Empty Rules		✓			41.245	41.245	4,02	4,02	1,75	317,26	34.910,82
Assignm. in Cond. Stat.			✓		77.438	77.438	7,55	7,55	2,47	179,95	34.983,37
Skipped Headings	✓				26.313	26.313	2,57	2,57	2,57	377,14	35.503,76
Long Parameter List			✓		61.321	61.321	5,98	5,98	1,96	227,24	44.178,05
Too General Selectors		✓			24.962	24.962	2,43	2,43	1,06	524,22	57.683,56
Complex Switch Case			✓		35.924	35.924	3,50	3,50	1,15	387,89	75.410,37
Repeated Switches			✓		29.697	29.697	2,90	2,90	0,95	469,23	91.222,75
Long Method			✓		28.028	28.028	2,73	2,73	0,89	497,17	96.654,85
Nested Callbacks			✓		24.784	24.784	2,42	2,42	0,79	562,24	109.306,09
Array Length Assignm.			✓		22.904	22.904	2,23	2,23	0,73	608,39	118.278,12
Deprecated Styl. Tags	I			✓	7.660	7.660	0,75	0,75	0,75	1.295,51	121.959,57
JS in HTML Links	I			✓	7.201	7.201	0,70	0,70	0,70	1.378,08	129.733,41
Uppercase Tags	✓				7.003	7.003	0,68	0,68	0,68	1.417,05	133.401,45
Chained Methods			✓		18.015	18.015	1,76	1,76	0,58	773,50	150.377,03
Deprecated Tags	✓				315	315	0,03	0,03	0,03	31.503,41	2,9657M

Tabelle 7.11.: Sprachübergreifende Verbreitung der einzelnen Smells

ISI: Identified Smell Instances, Kat.: Anzahl an Kategorien, $\frac{ISI}{Kat.}$: Gerundet auf ganze Zahl, SpW: Smells per Website, SpF: Smells per File, LpS: Lines per Smell, CpS: Characters per Smell, M: Millionen, I: Sprache, in der SoC-Smell identifiziert wurde und deren Werte für Datei-, Zeilen- und Zeichenanzahl angewendet werden

(1.304,89), *Too Long Rules* (1.393,39) und *Argument Count Mismatch* (1.406,22). Wiederum weiten Abstand hat *Mysterious Name* (2.080,89), welcher als einziger Smell hinsichtlich des $\frac{ISI}{Kat.}$-Wertes (570.455) zugunsten von *Too Much Cascading* (637.455) auch aus der „Top Ten“ fallen würde.

Bezüglich der 29 weiteren untersuchten Code Smells lässt sich ein fortlaufender und ziemlich gleichmäßiger Anstieg der *CpS*-Werte feststellen. Dabei gibt es lediglich einen extremen Sprung hinsichtlich der Werte – und der betrifft ausschließlich den letztplatzierten Smell *Deprecated Tags*. Er ist mit einem *CpS*-Wert von 2.965.747,05 extrem abgeschlagen und taucht 19,72-mal seltener auf als der vorletzte Smell *Chained Methods* (150.377,03).

Ein Blick auf die Verteilung der Smell-Kategorien visualisiert weitere interessante Erkenntnisse. So handelt es sich bei den „Top Vier“ jeweils um CSS-Smells. Gleichzeitig ist kein einziger CSS-Smell in den letzten zehn Positionen vertreten. Im oberen Bereich der Liste folgt auf die CSS-Smells ein Block von JavaScript-Smells auf den Positionen drei und fünf bis acht. Gleichzeitig fällt auf, dass kein *SoC*-Smell auf den obersten elf Positionen vertreten ist. Auch um die HTML-Smells steht es ähnlich: Während mit *Lengthy Lines* und *Mysterious Name* zwar zwei Smells in den „Top Ten“ vertreten sind, die in HTML identifiziert werden können, befindet sich der erste HTML-spezifische Smell jedoch erst auf Position 15 (*Opening Tag Missing Closing Tag*). Diese Erkenntnisse stützen die bisherige Ergebnis-Interpretation, wonach Code Smells insbesondere in CSS stark verbreitet sind und *SoC*-Smells seltener vorkommen als Smells in HTML und JavaScript.

Ein Ziel dieser Arbeit war es, herauszufinden, welche Smells besonders häufig verbreitet sind, damit diese in der Lehre und Fortbildung von Web-Entwickler*innen priorisiert behandelt werden können. Mit möglichst wenig Aufwand könnte somit mutmaßlich ein Großteil von Smell-Instanzen bereits in der Entstehung verhindert werden. Mit dem Vorliegen dieser Ergebnisse lässt sich diese These nun prüfen.

Unter der – für die Realität wohl etwas zu optimistischen – Annahme, dass Entwickler*innen die Entstehung von Code Smells vollständig verhindern würden, wenn sie ausreichend über diese aufgeklärt wären, hätten mit der Priorisierung der zehn häufigsten Smells bereits 90,91% (31,11 Mio. von 34,23 Mio.) der in der Stichprobe identifizierten Smell-Instanzen verhindert werden können. Durch die Priorisierung von nur fünf Smells wären immerhin noch 59,50% der Smells vermeidbar gewesen.

Eine weitere Kalkulation bestätigt für diese Hypothese das *Paretoprinzip* („80-20-Regel“), wonach 80% der Smell-Instanzen durch 20% der untersuchten Smells gedeckt werden müssten [DST14]: Die Analyse der obersten acht Smells (20,51% der Gesamtheit von 39 Smells) macht 81,94% aller identifizierten Smell-Instanzen aus.

Eine weitere Beobachtung stützt die Wichtigkeit der Priorisierung weniger Smells weiter. So fällt auf, dass Code Smells, die – nach subjektivem Empfinden – ziemlich bekannt zu sein scheinen und häufig als Beispiel für Code Smells genutzt werden, in Tabelle 7.11 auffallend tief aufzufinden sind. Beispiele hierfür sind *Commented-Out Code* (#24), *Long Parameter List* (#28), *Long Method* (#32) und *Chained Methods* (#38). Es entsteht der Ein-

druck, dass eine höhere Bekanntheit von Code Smells tatsächlich in einer geringeren Verbreitung resultiert. Um diese Hypothese jedoch mit wissenschaftlichen Mitteln zu prüfen, müsste eine Erhebung über die Bekanntheit von Code Smells unter Entwickler*innen durchgeführt werden. Deren Resultate könnten anschließend mit den Ergebnissen dieser Arbeit kombiniert werden.

Teilergebnis RQ.III)

Wie stark sind die einzelnen Code Smells verbreitet?

Die gesamten Ergebnisse des sprachübergreifenden Vergleichs der Verbreitung aller 39 analysierten Code Smells sind Tabelle 7.11 zu entnehmen.
Die zehn häufigsten Smells sind *Properties with Hard-Coded Values*, *High Specificity Values*, *Duplicated Code*, *Usage of !important*, *Excessive Global Variables*, *Depth*, *Lengthy Lines*, *Too Long Rules*, *Argument Count Mismatch* und *Mysterious Name*. Die zehn genannten Smells machen 90,91% aller Smell-Instanzen aus.
Zusätzlich wurde bewiesen, dass sich das *Paretoprinzip* ziemlich genau auf die Verbreitung von Code Smells anwenden lässt. Eine Priorisierung der häufigsten Smells in der Lehre ist somit höchst sinnvoll – die Motivation für diese Arbeit bewies sich als äußerst berechtigt.

7.4. Analyse des Zusammenhangs zwischen der Verbreitung und gemeinsamer Eigenschaften der untersuchten Smells

Zur Beantwortung von RQ.IV) wurden die Code Smells hinsichtlich verschiedener Eigenschaften mit Labels versehen. Anhand dieser Labels soll nun untersucht werden, ob Smells mit bestimmten Eigenschaften auffallend häufiger verbreitet sind als andere.

Die Resultate dieser Analyse sind in Tabelle 7.12 aufgeführt. Für den Vergleich wurde jeweils der Mittelwert aus den *SpW*-, *SpF*-, *LpS*- und *CpS*-Werten aller Smells mit dem jeweiligen Label gebildet. Als am aussagekräftigsten ist erneut der *CpS*-Wert zu sehen. Zusätzlich wurde der Variationskoeffizient bestimmt, um die Streuung um den Mittelwert einordnen zu können. Bei sehr hohen *CV*-Werten kann, unabhängig von den errechneten Mittelwerten, nicht von einer konstanten Verbreitung der Smells mit gleichen Eigenschaften die Rede sein.

Abermals zeichnet sich eine eindeutige Dominanz von Smells mit CSS-Bezug ab. Auf den obersten drei Positionen platzieren sich alle drei CSS-spezifischen Labels. Trotz der generell hohen Verbreitung beider Eigenschaften zeigt sich ein extrem deutlicher Unterschied zwischen Smells, die in CSS-Deklarationen und -Selektoren auftreten. Deklaration-Smells sind mit durchschnittlich einer Instanz in 827,20 Zeichen 21,22-mal häufiger vertreten als Selektor-Smells (alle 17.551,50 Zeichen). Der Wert für Deklaration grenzt sich somit extrem von allen anderen untersuchten Labels ab. Zusätzlich sticht Deklaration

Label	NoAS	Wert ⇒	SpW	SpF	LpS	CpS
Deklaration	3	Mittelwert	370,81	126,45	6,73	827,20
		CV in %	47,64	71,38	48,05	53,32
Selektor	5	Mittelwert	93,47	40,77	159,51	17.551,50
		CV in %	155,98	155,98	118,85	118,85
CSS	16	Mittelwert	146,95	48,97	187,40	19.859,26
		CV in %	140,08	142,43	170,46	154,97
Überflüssig	6	Mittelwert	106,58	20,75	214,02	32.864,89
		CV in %	205,58	195,75	70,31	86,64
Fehlerähnlich	5	Mittelwert	96,41	32,84	180,54	33.049,81
		CV in %	117,86	110,56	123,87	134,28
SoC	8	Mittelwert	13,30	8,28	426,11	41.351,41
		CV in %	100,30	102,32	124,93	119,20
JavaScript	22	Mittelwert	90,04	22,90	273,38	44.366,27
		CV in %	170,84	149,18	121,03	108,77
Optisch	13	Mittelwert	109,66	25,67	262,74	47.050,13
		CV in %	165,41	142,41	94,69	104,75
Inhaltlich	4	Mittelwert	4,79	1,82	426,09	79.854,34
		CV in %	60,98	73,43	59,82	66,41
HTML	15	Mittelwert	40,63	11,57	2.479,08	234.351,32
		CV in %	233,18	139,73	313,53	312,12
Veraltet	3	Mittelwert	2,13	2,13	10.990,33	1,0346M
		CV in %	116,46	116,46	132,05	132,05

Tabelle 7.12.: Mittelwerte der ermittelten Verbreitungswerte aller Smells eines Labels

NoAS: Number of Analyzed Smells, SpW: Smells per Website, SpF: Smells per File, LpS: Lines per Smell, CpS: Characters per Smell, M: Millionen, CV: Variationskoeffizient

mit dem niedrigsten Variationskoeffizienten aller Labels hervor (53,32%). Code Smells in CSS-Deklarationen sind also nicht nur im Mittel extrem stark verbreitet, sondern befinden sich auch konstant über die unterschiedlichen Smells hinweg auf diesem hohen Niveau. Bei dieser Beobachtung sollte jedoch der recht geringe *NoAS*-Wert beachtet werden – das Ergebnis stützt sich also auf eine recht kleine Stichprobe. Dennoch zeigt sich als ein Ergebnis dieser Arbeit, dass CSS-Deklarationen extrem anfällig für das Einbringen von Code Smells sind.

HTML, CSS und JavaScript unterscheiden sich von den bisherigen kategoriebasierten Ergebnissen darin, dass auch *SoC*-Smells mit jeweiligem Sprachbezug mit dem Label versehen wurden. Auch in dieser Betrachtung ist ein durchschnittlicher CSS-Smell deutlich öfter vertreten als jener der anderen Sprachen. CSS positioniert sich an dritter Stelle

mit einem *CpS*-Wert von 154,97. Außerdem zeigt sich, dass durchschnittliche JavaScript- und HTML-Smells bei dieser Betrachtung sogar seltener vertreten sind als *SoC*-Smells: SoC (alle 41.351,41 Zeichen) ordnet sich auf Rang sechs ein, gefolgt von JavaScript (alle 44.366,27 Zeichen) auf Position sieben und HTML an zehnter und somit vorletzter Stelle (alle 234.351,32 Zeichen).

Insbesondere die Positionierung von HTML relativiert sich jedoch deutlich durch den höchsten CV-Wert von 312,12%. Er zeigt, dass sich die einzelnen Werte sehr um den *CpS*-Mittelwert streuen. Ein Blick auf die Verbreitung der einzelnen Smells (⇒ Tabelle 7.11) offenbart die Ursache der hohen *CpS*- und *CV*-Werte für HTML: Sie werden durch den mit Abstand seltensten Smell *Deprecated Tags* stark nach oben getrieben.

Die restlichen fünf Labels beziehen sich weder auf bestimmte Stellen im Code, in denen der Smell auftritt, noch auf dessen Sprache. Hingegen handelt es sich bei Überflüssig, Fehlerähnlich, Optisch, Inhaltlich und Veraltet um inhaltliche Eigenschaften der Code Smells.

Eine sehr ähnliche Verbreitung hinsichtlich des *CpS*-Wertes ergibt sich für Smells mit den Labels Überflüssig (32.864,89) und Fehlerähnlich (33.049,81) auf den Positionen vier und fünf. Ein großer Anteil von Code Smells im Web basiert somit auf redundantem oder unnötigem Code. Zusätzlich zeigt sich anhand des vergleichsweise niedrigen Variationskoeffizienten (86,64%), dass die verschiedenen Smells mit der Eigenschaft Überflüssig eine konstant hohe Verbreitung aufweisen. Die hohe Verbreitung dieser Smells dürfte insbesondere in hohem Aufwand bei der Wartung und Weiterentwicklung resultieren und kann Web-Projekte künstlich aufblähen und den Code somit unübersichtlicher machen. Auch die hohe Verbreitung von Smells mit der Eigenschaft Fehlerähnlich ist bedenklich. Diese Smells liegen inhaltlich bereits sehr nahe an konkreten Fehlern und können diese verursachen. Anders als bei anderen Smells resultieren hier somit nicht nur langfristige Wartungsprobleme, sondern bereits kurzfristige Störungen des gewünschten Programmablaufs oder der optischen Erscheinung einer Website. Der etwas höhere Variationskoeffizient (134,28%) zeigt jedoch, dass nicht jeder Smell mit dieser Eigenschaft eine hohe Verbreitung aufweist.

Smells mit dem Label Optisch erhöhen die Komplexität von Code durch eine unübersichtliche Formatierung oder Strukturierung. Mit einem *CpS*-Wert von 47.050,13 positioniert sich ein durchschnittlicher Smell dieser Kategorie an Stelle sieben und somit im unteren Mittelfeld. Auch die Streuung um diesen Mittelwert ist weder auffallend hoch noch auffallend niedrig (104,75%).

Smells, die Code unverständlicher machen, da sich für eine inhaltlich unnötig komplexe Lösung entschieden wurde, wurden mit dem Label Inhaltlich versehen. Ein Smell mit dieser Eigenschaft tritt im Mittel nur alle 79.854,34 Zeichen auf. Inhaltlich schlechte Umsetzungen scheinen somit keinen hohen Anteil an der Gesamtheit der Komplexitätstreiber in der Web-Entwicklung zu haben. Der niedrige Variationskoeffizient (66,41%) bestätigt außerdem, dass sich diese Aussage auch auf die einzelnen Smells mit dieser

Eigenschaft übertragen lässt.

Das Schlusslicht in dieser Betrachtung bildet das Label **Veraltet**. Anhand des mit Abstand höchsten *CpS*-Wertes von 1,0346M wird deutlich, dass die Nutzung von Elementen, die als „deprecated“ eingestuft wurden, sehr selten stattfindet – zumindest im Vergleich zum Einbringen anderer Formen von Code Smells.

Teilergebnis RQ.IV)

Weisen Code Smells mit bestimmten Eigenschaften eine höhere Verbreitung auf als andere?

Ja, es konnten – teilweise deutliche – Unterschiede in der Verbreitung von Smells mit verschiedenen Eigenschaften festgestellt werden.
Abermals zeigte sich eine deutliche Dominanz von Code Smells, die mit der Sprache CSS in Zusammenhang stehen. Dabei sind Smells innerhalb von CSS-Deklarationen nochmals um ein vielfaches häufiger als in Selektoren.
Auch Code Smells, die auf unnötigem oder redundantem Code basieren oder aber inhaltlich sehr nah an konkreten Programmfehlern liegen, weisen eine hohe Verbreitung auf.
Weniger häufig sind Code Smells, die den Code für Entwickler*innen optisch unübersichtlicher machen oder die Komplexität durch inhaltlich schlechte Lösungen erhöhen. Die mit Abstand geringste Verbreitung wurde für die Nutzung von veralteten Code-Elementen festgestellt.

Bei der Analyse der Verbreitung einzelner Smells in den jeweiligen Kategorien (⇒ Kapitel 7.3) wurde teilweise eine inhaltliche Gemeinsamkeit zwischen Smells mit einem besonders hohen Variationskoeffizienten festgestellt. Diese diagnostizierte hohe Streuung um den Verbreitungsmittelwert je Website spricht für ein Vorkommen nach dem Muster „*Ganz oder gar nicht*“ – diese Smells treten in den meisten Fällen also entweder sehr unter- oder sehr überdurchschnittlich häufig auf den einzelnen Websites auf. Die bereits zugeordneten Labels ermöglichen eine nähere Analyse der ersten Beobachtungen, auf Basis des vorhandenen Datensatzes. Gibt es einen Zusammenhang zwischen bestimmten Eigenschaften von Code Smells und ihrer „*Ganz oder gar nicht*“-Verbreitung auf den einzelnen Websites?

Um dieser Frage nachgehen zu können, wird ein Streuungsmittelwert für die einzelnen Labels berechnet. Dabei handelt es sich um den Mittelwert der *Je Website*-Variationskoeffizienten (⇒ Tabellen der Kapitel 7.3.1 - 7.3.4) aller Smells mit dem jeweiligen Label. Da es sich bei dem Variationskoeffizienten um ein relatives Streuungsmaß handelt, welches den Vergleich von Streuungen bei unterschiedlichen Stichprobengrößen ermöglicht, ist auch die Bildung eines Mittelwertes mathematisch legitimiert und führt zu aussagekräftigen Ergebnissen. Für Smells, welche in mehreren Sprachen analysiert

wurden, wird vorab bereits ein sprachübergreifender Streuungsmittelwert berechnet. Somit zählen diese Smells nicht doppelt oder dreifach in den Label-Streuungsmittelwert. Aus der Streuungsmittelwertberechnung resultiert abermals ein Variationskoeffizient, der somit angibt, wie konstant die einzelnen Smells eines Labels die ermittelte Streuung repräsentieren.

Label	Streuungsmittelwert	Variationskoeffizient
Deklaration	263,76%	19,20%
Selektor	274,73%	33,80%
Inhaltlich	294,43%	15,18%
JavaScript	323,41%	65,37%
Überflüssig	375,00%	32,14%
Optisch	403,39%	67,55%
Fehlerähnlich	410,48%	106,81%
CSS	450,49%	123,42%
SoC	709,45%	107,38%
HTML	1.091,18%	112,42%
Veraltet	2.784,16%	56,50%

Tabelle 7.13.: Streuung je Website der Smells mit bestimmten Eigenschaften

Die Ergebnisse der beschriebenen Berechnungen sind Tabelle 7.13 zu entnehmen. Dabei resultieren niedrigere Streuungsmittelwerte in der Erkenntnis, dass die Smells mit der jeweiligen Eigenschaft recht nah um den Verbreitungsmittelwert verteilt sind. Besonders interessant sind hingegen hohe Werte, welche die beschriebene Verbreitung der Art „*Ganz oder gar nicht*“ belegen.

Die Smells mit den Labels Deklaration, Selektor und Inhaltlich weisen die niedrigsten Streuungsmittelwerte auf (260% - 300%). Sie sind somit recht nah um den Verbreitungsmittelwert verteilt. Die ebenfalls niedrigen Variationskoeffizienten zeigen, dass diese Beobachtung auch sehr konstant für die einzelnen Smells gilt.

Die Labels JavaScript, Überflüssig, Optisch, Fehlerähnlich und CSS weisen Streuungsmittelwerte im Bereich von 320% bis etwa 450% auf. Die Verbreitung dieser Smells auf den einzelnen Websites weicht somit etwas stärker von dem errechneten Verbreitungsmittelwert ab. Insbesondere für die Labels CSS und Fehlerähnlich ergeben sich auch erhöhte – jedoch nicht extreme – Variationskoeffizienten. Somit ist die erhöhte Streuung nicht auf alle Smells mit diesem Label übertragbar. Die Vermutung im Rahmen von Kapitel 7.3.3, wonach Smells im Themenbereich Code-Wiederholung entweder sehr

häufig oder sehr selten je Website auftreten, bestätigte sich nicht. Das zugehörige Label Überflüssig weist keinen extrem hohen Streuungsmittelwert auf.

Hingegen lassen sich für die Labels SoC (709,45%) und HTML (1.091,18%) auffallend hohe Streuungsmittelwerte feststellen, welche auch aus rein logischer Sicht gut erklärbar sind. Entweder sind sich Entwickler*innen der korrekten *Separation of Concerns* bewusst und halten sich konsequent daran – oder eben nicht und sie verstoßen häufiger dagegen. Dieser Effekt überträgt sich auch auf die *SoC*-Smells, die mit HTML versehen wurden. Darüber hinaus lässt sich für weitere HTML-Smells wie *Deprecated Tags* und *Deprecated Attributes* eine starke Abhängigkeit vom Entwickler*innen-Bewusstsein vermuten. Auch für *Uppercase Tags* liegt der „*Ganz oder gar nicht*"-Charakter sehr nahe: Tags werden konstant entweder groß oder klein geschrieben, eine Durchmischung ergibt wenig Sinn.

Anders als die in Kapitel 7.3.3 aufgestellte Vermutung hat sich jene aus Kapitel 7.3.1 eindeutig bestätigt. Im Rahmen dieser Analyse hat sich für die Gesamtheit der untersuchten Veraltet-Smells, ein extremer Streuungsmittelwert von 2.874,16% ergeben. Zusätzlich liegt der Variationskoeffizient mit 56,50% relativ niedrig. Somit ergibt sich aus dem Datensatz der Charakter einer deutlichen „*Ganz oder gar nicht*"-Verbreitung für Smells, die die Nutzung veralteter Elemente thematisieren. Diese ist zusätzlich auf die einzelnen Smells mit dieser Eigenschaft übertragbar. Neben dem zuvor beschriebenen Aspekt des Entwickler*innen-Bewusstseins ist der hohe Wert vermutlich auch auf die Analyse einiger sehr veralteter und nicht aktualisierter Websites zurückzuführen.

Insgesamt ergaben sich hohe Streuungsmittelwerte für jene Labels, bei denen diese auch logisch gut erklärbar sind. Somit bestätigt die getätigte Analyse, dass diese theoretischen Überlegungen auch konkrete praktische Auswirkungen haben. Wie auch bei der Verbreitung ist bei der Interpretation der Ergebnisse zu beachten, dass manche Labels nur durch wenige Smells repräsentiert wurden. Eine allgemeine Übertragbarkeit der Erkenntnisse könnte daher an Verfälschungen durch niedrige Stichproben scheitern.

Zusätzliches Ergebnis

Die Daten zeigen einen Zusammenhang zwischen bestimmten Smell-Eigenschaften und einer entweder sehr über- oder unterdurchschnittlichen Verbreitung auf einzelnen Websites. Demnach besitzen vor allem Smells im Kontext der *Separation of Concerns*, die mit HTML im Zusammenhang stehen oder die Nutzung veralteter Elemente thematisieren, den Charakter einer „*Ganz oder gar nicht*"-Verbreitung.

8. Kritische Reflexion der Validität

8.1. Methodischer Ansatz

Im Zuge dieses Kapitels werden das Vorgehen und die Ergebnisse dieser Arbeit sowie deren Interpretation einer kritischen Reflexion unterzogen. Dafür werden vier Kategorien von möglichen *Threats*, also potenziellen Bedrohungen hinsichtlich der Validität, untersucht: *Construct Validity Threats* (⇒ Kapitel 8.2), *Internal Validity Threats* (⇒ Kapitel 8.3) *External Validity Threats* (⇒ Kapitel 8.4) und *Reliability Threats* (⇒ Kapitel 8.5). Bei der folgenden Reflexion wird sich an den Beschreibungen der jeweiligen Threat-Kategorien von Robert K. Yin, im Rahmen des Werkes *Case Study Research and Applications – Design and Methods* [Yin18] sowie deren Interpretation in anderen Arbeiten [BR16, RR22, KDPG09, JKA19], orientiert.

8.2. Construct Validity Threats

Bei *Construct Validity Threats* handelt es sich um Bedrohungen hinsichtlich der Relation zwischen der Theorie und den Beobachtungen. Insbesondere geht es dabei um die Frage, ob die korrekten Maße für die untersuchten Konzepte gefunden und angewendet wurden sowie um die Angemessenheit der gewählten Methoden.

Ziel dieser Arbeit war es, die Verbreitung verschiedener Code Smells im Web zu analysieren. Das entscheidende und einzig logisch anzuwendende Maß für diese Analyse ist die Häufigkeit des Auftretens eines Smells. Interessant wird es daher insbesondere bei der näheren Betrachtung des Kontextes, in dem dieses Maß Anwendung fand.

Bei der untersuchten Stichprobengröße liegt die einzige Möglichkeit, diese Analyse zu ermöglichen, in der automatisierten Erkennung der Code Smells. Daraus resultiert eine starke Abhängigkeit zwischen den Idenfikationsalgorithmen und den resultierenden Ergebnissen. Es muss davon ausgegangen werden, dass es bei der Identifikation der Smell-Instanzen *False Positives* und *False Negatives* gab und, aufgrund der unterschiedlichen Algorithmen, deren Anzahl von Smell zu Smell variieren könnte. Gleichzeitig kann dadurch, dass es sich bei dem angewandten Programm *WebDeo* um eine Eigenentwicklung handelt, welche durch einen einzelnen Entwickler entstand, von einer relativ konstanten Qualität der Identifikationsalgorithmen ausgegangen werden. Es sind somit keine gravierenden Unterschiede zu erwarten, welche etwa aufkommen könnten, wenn manche Smells mit einem sehr unprofessionellen und andere mit einem extrem professionellen, kommerziellen Tool identifiziert worden wären.

Außerdem ist zu erwähnen, dass keine Ansätze der Erkennung von Code Smells vermischt wurden. Alle 39 untersuchten Smells wurden auf Basis der statischen Code-Ana-

lyse und ihrer Grenzen und Möglichkeiten identifiziert. Ergebnis-Verfälschungen, aufgrund der Unterschiede in der Genauigkeit von grundlegenden Smell-Identifkationsansätzen, sind somit ausgeschlossen.

Eine sehr große Abhängigkeit der Ergebnisse besteht auch zu den gewählten Smell-Metriken und den zugehörigen Schwellenwerten. Aufgrund dieses vorhersehbar großen Einflusses wurde sich von vornherein dazu entschieden, keine eigenen Metriken oder Schwellenwerte zu wählen. Hingegen ist jede angewandte Metrik in anderen wissenschaftlichen Arbeiten aufzufinden. Hinsichtlich der genutzten Schwellenwerte ist umfassend in wissenschaftlichen Werken recherchiert worden. Wurden dabei unterschiedliche Angaben entdeckt, wurde der Mittelwert aus den gelisteten Schwellenwerten gebildet. Die Quellen für die angewandten Schwellenwerte wurden stets referenziert und somit transparent kommuniziert. Die Wahl sämtlicher Metriken und Schwellenwerte, die in dieser Arbeit genutzt wurden, ist somit von persönlichen Einflüssen verschont geblieben und bewegt sich vollständig im Rahmen des bestehenden wissenschaftlichen Konsens.

Etwas anders sieht es bei *booleschen* Smells aus – also jenen Code Smells, die unabhängig von Schwellenwerten, entweder auftreten oder nicht. Bei diesen Smells wurde umfassend über die verschiedenen Fälle des Auftretens recherchiert. Mithilfe von Unit-Tests wurde abgesichert, dass alle identifizierten Fälle von *WebDeo* erkannt werden können, um eine möglichst gleichmäßige und hohe Identifikationsrate sicherzustellen. Gleichzeitig ist das Übersehen weiterer Fälle nicht ausgeschlossen und konnte auch kaum verhindert werden. Die Komplexität dieser Problematik wurde im Rahmen von Kapitel 6.3.4 anhand des Smells *Array Length Assignment* thematisiert.

Ein weiteres relevantes Maß ist der Umfang der analysierten Stichprobe. Diese liegt mit 10.255 analysierten Websites, über 65.000 Dateien und etwa 37 Mio. Code-Zeilen deutlich höher als in vergleichbaren Arbeiten (⇒ Kapitel 7.1) und ist daher auch als überdurchschnittlich aussagekräftig zu werten. In Anbetracht der gewaltigen Größe des World Wide Webs ist dennoch zu hinterfragen, inwiefern die Stichprobe als repräsentativ angesehen werden kann.

Auch die Anzahl von 39 untersuchten Smells ist viel höher als in vergleichbaren Arbeiten (⇒ Kapitel 3) und den meisten Tools (⇒ Kapitel 3.2). Für manche der getätigten Analysen ist diese Auswahl jedoch noch immer relativ gering. So wurden manche Smell-Eigenschaften – untersucht anhand der verschiedenen Labels – lediglich von drei Smells repräsentiert. In Anbetracht des vorgegebenen Umfangrahmens für diese Arbeit und des Fakts, dass es sich um ein Ein-Personen-Projekt handelt, ist die Anzahl von 39 untersuchten Smells jedoch als sehr hoch einzuschätzen. Aufgrund des Entwicklungsaufwands, der für die Identifikation jedes der 39 Smells anfiel, hätte eine noch größere Auswahl diesen Rahmen gesprengt.

Zu betrachten ist allerdings, welche Auswahl an Smells letztlich getroffen wurde. Als Maß hierfür wurde die *Relevanz in der Web-Entwicklung* herangezogen – eine Phrase, welche sich sogar im Titel der Arbeit widerspiegelt. Die Entscheidung darüber, dass die 39

untersuchten Smells „relevanter“ und geeigneter seien als die 27 zwar betrachteten, jedoch aussortierten Smells, war rein subjektiv. Somit handelt es sich um um eine Einflussnahme auf den Untersuchungsgegenstand der Arbeit. Gleichzeitig ist vollkommen klar, dass eine solche Analyse nicht ohne das Fällen von individuellen Entscheidungen durchgeführt werden kann. Ebenso kann nicht jede dieser Entscheidungen auf messbaren, rationalen und objektiven Standpunkten basieren. Außerdem wurde die Aussortierung der 27 Smells begründet festgehalten (⇒ Tabelle 5.1) und damit versucht, die Entscheidung so nachvollziehbar wie möglich zu gestalten.

Zur Veranschaulichung der Ergebnisse wurden verschiedene Maße eingesetzt (*ISI*, *SpW*, *SpF*, *LpS*, *CpS*, ...). Somit wird eine differenzierte Betrachtung der Ergebnisse aus verschiedenen Sichtweisen ermöglicht. Die potenziellen Nachteile der Festlegung auf eine einzelne Kennzahl wurden damit vermieden. Gleichzeitig wurden bei der textlichen Interpretation gelegentlich bestimmte Werte priorisiert, beispielsweise der *Characters per Smell*-Wert. Gründe für diese Priorisierung wurden stets nachvollziehbar dargelegt. Durch die Vollständigkeit der Tabellen werden Leser*innen der Arbeit eigene Interpretationen anhand anderer Kennzahlen ermöglicht.

An angemessenen Stellen wurden die eigenen Maße zusätzlich um anerkannte mathematische Werte, wie relative Anteile, Mittelwerte, Variationskoeffizienten und Konfidenzintervalle, ergänzt. Sie verbessern die Möglichkeiten zur Einordnung der Ergebnisse und bergen durch die ihre breite Nutzung und Bekanntheit weniger Risiken im Vergleich zu selbst aufgestellten Maßen.

Hinsichtlich der Angemessenheit der angewandten Methoden sind noch drei grundlegende Entscheidungen zu reflektieren. Das strenge Limit von nur einer analysierten Web Page je Domain (E1), die zufällige Auswahl von Start-URLs für den Crawler (E2) und das Einbringen eines Framework-Filters (E4) wirkte sich jeweils auf den resultierenden Datensatz aus. Im Rahmen der Kapitel 6.1 und 6.3.3 wurden all diese Entscheidungen ausführlich begründet. Auf Basis dieser Begründungen wird davon ausgegangen, dass sich all diese Entscheidung – im Vergleich zu potenziellen Alternativen – positiv auf die Qualität des Datensatzes und der Ergebnisse ausgewirkt haben.

8.3. Internal Validity Threats

Internal Validity Threats beschreiben Bedrohungen der internen Validität der Ergebnisse durch unerwartete oder akzeptierte externe Einflüsse.

Die theoretischen Grundlagen dieser Arbeit basieren – wann immer dies möglich war – auf einem Konsens aus mehreren wissenschaftlichen Quellen. Somit können negative Einflüsse auf Basis von möglichen Falschinformationen reduziert werden. Darüber hinaus wurde sehr kritisch und transparent mit fragwürdigen Informationen umgegangen. Beispiele hierfür finden sich in Kapitel 5.1. Dort wurden die Beschreibungen der angeblichen Code Smells *Erroneous Adjoining Selectors* und *Undoing Styles* auf Basis anderer

Quellen kritisch hinterfragt und im Resultat nicht in den weiteren Arbeitsschritten berücksichtigt. Die Übernahme falscher Informationen oder eine fehlerhafte Interpretation von Informationen aus externen Quellen kann dennoch nie vollständig ausgeschlossen werden. Durch den sehr bewussten und kritischen Umgang mit den genutzten Quellen wurden Bedrohungen dieser Art jedoch nach bestem Wissen und Gewissen reduziert.

Weitere Abhängigkeiten bestehen zu den genutzten Software-Modulen, welche für die Erstellung der Code-Modelle genutzt wurden. Mit *BeautifulSoup*, *tinycss2* und einem Python-Port von *Esprima* wurde sich für weit verbreitete Open-Source-Module entschieden, die seit vielen Jahren genutzt und optimiert werden. Entsprechend kann von einer sehr geringen Fehlerquote ausgegangen werden, welche somit auch nur geringen Einfluss auf die interne Qualität dieser Arbeit haben dürfte.

Als ein weiterer potenzieller Einfluss auf den Datensatz kann der verhinderte Zugriff auf einige Websites durch den Web Crawler oder *WebDeo* angesehen werden. Grund für den scheiternden Zugriff waren etwa DDoS-Abwehrprogramme oder Server-Ausfälle zur Zeit des versuchten Zugriffs. Da es sich bei den Websites jedoch nicht um eine gezielte, vermeintlich repräsentative Auswahl handelte, ergeben sich aus den gescheiterten Zugriffen keine nennenswerten Folgen für den resultierenden Datensatz. Die Aussagekraft der Daten liegt in der hohen Stichprobe begründet, welche trotz der wenigen gescheiterten Zugriffe erzielt werden konnte.

8.4. External Validity Threats

External Validity Threats thematisieren die Generalisierbarkeit der Erkenntnisse dieser Arbeit. Dies ist – soweit dies auf Basis der getätigten Literaturrecherche bekannt ist – die bisher wohl umfassendste getätigte Analyse hinsichtlich der Verbreitung von Code Smells (⇒ Kapitel 7.1). Dies spricht grundlegend auch für eine hohe Generalisierbarkeit der Ergebnisse.

Dennoch handelt es sich spezifisch um eine Analyse im Kontext der Web-Entwicklung mit all ihrer Besonderheiten. Dieses Thema der Arbeit wurde bewusst gewählt und fortlaufend kommuniziert. Es kann aufgrund des spezifischen Kontextes jedoch nicht von einer Übertragbarkeit der Ergebnisse, etwa auf andere Programmiersprachen, ausgegangen werden.

Hinsichtlich der getätigten Vermutungen bezüglich der Ursache von Ergebnissen und den aufgestellten Hypothesen, wären weitere Analysen notwendig. Dies gilt auch hinsichtlich der allgemeinen Ergebnisse, welche sich durch Analysen in einem anderen Projektumfeld – etwa unter Einsatz eines anderen Smell-Identifikationstools – bestätigen ließen.

Insbesondere bei Ergebnissen, die auf einer kleineren Stichprobe basieren, ist die Generalisierbarkeit kritisch zu sehen. Ein bereits thematisiertes Beispiel hierfür sind die Erkenntnisse im Rahmen von Kapitel 7.4 für Labels, welche nur von wenigen Smells re-

präsentiert wurden. An diesen Stellen wurde diese Schwäche, in der Interpretation der Ergebnisse, stets transparent hervorgehoben.

Generell gilt: Obwohl von einer grundlegenden Übertragbarkeit der zentralen Erkenntnisse dieser Arbeit ausgegangen werden kann, sollte der Kontext, in dem die Ergebnisse erzielt wurden, bei der Interpretation aller Ergebnisse stets berücksichtigt werden.

8.5. Reliability Threats

Reliability Threats thematisieren Bedrohungen hinsichtlich der Möglichkeit zur Wiederholung einer Arbeit.

Bei einer wiederholten Durchführung der getätigten Analyse würde man nicht die exakt gleichen Daten erhalten, welche im Rahmen der Ergebnisse präsentiert wurden. Grund hierfür ist, dass es sich bei dem World Wide Web um eine sich ständig verändernde Datenbasis handelt. Beim Einsatz des Web Crawlers würden andere Websites gesammelt werden. Selbst bei einer Durchführung auf dem gleichen Satz an Websites würden für die Verbreitung der Code Smells andere Ergebnisse erzielt werden, da einige der Websites in der Zwischenzeit aktualisiert worden wären. Gleichzeitig ist durch die Größe der analysierten Stichprobe davon auszugehen, dass auch bei der Analyse anderer oder überarbeiteter Websites, in den grundlegenden Erkenntnissen übereinstimmende Ergebnisse erzielt werden würden – lediglich die exakten Zahlen dürften leicht variieren.

Die Möglichkeit zur Wiederholung der durchgeführten Arbeit wurde während des gesamten Prozesses bedacht und gefördert. Sämtliche getätigte Schritte und Entscheidungen sind in der schriftlichen Ausarbeitung nachvollziehbar dargelegt. Sämtliche Komponenten aller genutzten Tools – inklusive dem kompletten Quellcode von *WebDeo*, dem Web Crawler und den genutzten Konvertierungsskripten zum Ermöglichen der Daten-Analyse – sind in einem Repository auf den Servern der Universität Hamburg öffentlich verfügbar [Sch]. Das Gleiche gilt auch für den exakten Datensatz der präsentierten Analyse, welcher sowohl in Form der Rohdaten als auch in der getätigten Auswertung veröffentlicht wurde. Auch die komplette Entwicklungs- und Datenhistorie ist mit allen Commits in dem Repository exakt nachvollziehbar. Es herrscht somit eine sehr hohe Transparenz hinsichtlich der durchgeführten Arbeitsschritte. Sowohl die Analyse der getätigten Vorgänge als auch die Möglichkeit das Experiment unter gleichen Bedingungen zu wiederholen, wurde somit aktiv ermöglicht und gefördert.

9. Fazit und Ausblick

In dieser Arbeit wurde die Verbreitung von 39 Code Smells, welche für die Web-Entwicklung relevant sind, auf 10.255 Websites untersucht. Mit der Analyse von Smells in den Kategorien HTML, CSS, JavaScript und deren Zuständigkeitstrennung wurde das breite Spektrum der Web-Entwicklung bei dieser Analyse abgedeckt.

Mit *WebDeo* wurde ein neues Tool entwickelt, welches in der Lage ist, diese Smells nach der Angabe von URLs auf den dahinter liegenden Websites zu identifizieren. Durch die Bereitstellung eines Feedback-Reports ermöglicht *WebDeo* es Web-Entwickler*innen, die Instanzen der Code-Smells zu lokalisieren und unterstützt sie durch die Angabe sinnvoller Refactorings bei der Überarbeitung ihres Codes. Durch einen Vergleich mit der durchschnittlichen Verbreitung der jeweiligen Smells werden Nutzer*innen von *WebDeo* darüber aufgeklärt, welche Smells sie besonders häufig oder selten in ihren Code einbringen. Dieses Wissen bietet die Grundlage dafür, gezielt an persönlichen Schwächen arbeiten zu können und individuelle Stärken zu kennen. Durch die Entwicklung von *WebDeo* konnte das Ziel erreicht werden, ein Tool für Web-Entwickler*innen bereitzustellen, welches die Code-Qualität im Web in ihrer Gesamtheit betrachtet und überdurchschnittlich viele Smells identifizieren kann. Es besitzt das Potenzial, die Code-Qualität von den Nutzer*innen sowohl unmittelbar als auch langfristig verbessern zu können.

Gleichzeitig wären vielseitige Erweiterungen an *WebDeo* möglich. Um das Potenzial für den praktischen Einsatz des Tools auch außerhalb des wissenschaftlichen Kontextes auszuschöpfen, böte sich etwa die Übersetzung des Tools in die englische Sprache an. Somit könnte das Tool einem deutlich größeren Anteil an Nutzer*innen zugänglich gemacht werden. Die Entwicklung einer grafischen Benutzeroberfläche würde den Einsatz von *WebDeo* erleichtern. Hinsichtlich der inhaltlichen Erweiterung ist die Software-Architektur so konstruiert, dass ohne große Umstände die Erkennung für weitere Code Smells zu *WebDeo* hinzugefügt werden könnte. Mit der Möglichkeit, Schwellenwerte für die Smells selbst bestimmen zu können, ließen sich die Ergebnisse von *WebDeo* einfacher mit denen anderer Tools vergleichen und kombinieren. Die in dieser Arbeit getätigte Analyse hinsichtlich bestimmter Smell-Eigenschaften in der Form von Labels ließe sich ebenfalls in den Feedback-Bericht des Tools integrieren. Die Bewertung der Qualität der Smell-Identifikation von *WebDeo* hätte den Rahmen dieser Arbeit deutlich gesprengt – wäre zur weiteren Einordnung der Ergebnisse und zur Verbesserung des Tools jedoch sehr sinnvoll. Hierfür könnte ein Abgleich der Ergebnisse der automatisierten Erkennung von *WebDeo* mit der manuellen Identifikation von Code Smells durch erfahrene Entwickler*innen auf einem identischen Datensatz erfolgen.

Durch die auf Literaturrecherche basierende und mithilfe von *WebDeo* getätigte Analyse konnte ein umfangreicher Datensatz über die Verbreitung der Code Smells im Web

zusammengetragen werden. Dessen Auswertung resultierte in Ergebnisse für alle gestellten Forschungsfragen.

In dieser Arbeit wird eine Auswahl von 39 Code Smells vorgestellt, welche für die Web-Entwicklung als relevant eingeschätzt wurden und sich darüber hinaus mit den Mitteln der statischen Code-Analyse automatisiert identifizieren lassen. Dabei wurde deutlich, dass sich die klassischsten und bekanntesten Code Smells häufiger auf JavaScript als auf HTML und CSS übertragen lassen. Gleichzeitig zeigte sich, dass es HTML- und CSS-spezifische Smells gibt und sich das Konzept von Code Smells sehr gut auf die Strukturierungs- und Designsprachen übertragen lässt. Mit den Smells, die Verstöße gegen die *Separation of Concerns* thematisieren, wurde zusätzlich ein Set von Code Smells präsentiert und analysiert, welches sich aus der Besonderheit der Sprachkombination im Web ergibt.

Es konnte eine sehr hohe Verbreitung der 39 untersuchten Smells im World Wide Web nachgewiesen werden. Je untersuchter Website wurden sie durchschnittlich 3.337,53-mal identifiziert – je Smell ergibt das im Mittel 74,17 Instanzen. Hinsichtlich der untersuchten Sprachen konnten deutliche Unterschiede in der Häufigkeit von Code Smells ausgemacht werden. Mit Abstand am häufigsten wurden Code Smells in CSS identifiziert. Gemessen an der durchschnittlichen Zeichen-Anzahl je Smell-Instanz sind Code Smells auch in HTML häufiger anzutreffen als in JavaScript. Es zeigt sich somit, dass es hinsichtlich der Präsenz von Code Smells absolut keine Berechtigung für die diagnostizierte Forschungslücke gibt. Code Smells spielen in JavaScript eine sehr große Rolle (⇒ Kapitel 2.4), welche aber nicht im Ignorieren der Thematik in HTML und CSS resultieren darf. Der Fokus der Wissenschaft und von Tools auf einzelne Sprachen sowie die besondere Konzentration auf JavaScript-Smells sollte daher überwunden und die Web-Entwicklung in ihrer Gesamtheit betrachtet werden. Wenngleich es sich bei HTML und CSS nicht um Programmiersprachen handelt, kommt der Code-Qualität darin eine große Bedeutung zu. Das Konzept von Code Smells lässt sich jeweils sehr gut übertragen. Die höhere analysierte Verbreitung der Smells macht deutlich, dass dies auch viel häufiger passieren sollte.

Auch die *Separation of Concerns*-Smells haben einen relevanten Anteil an der Gesamtheit der identifizierten Web-Smells. Wenngleich dieser geringer ausfällt als die Anteile der jeweiligen Sprachen, ist er groß genug, um die weitere Erforschung hinsichtlich Smells dieser Kategorie zu empfehlen. Die Konzepte der engen Sprachkombination sind eine Besonderheit der Web-Entwicklung. Bei einer ganzheitlichen Betrachtung des Themas Code-Qualität sollten sie daher auch berücksichtigt werden.

Aus der insgesamt sehr hohen nachgewiesenen Verbreitung resultiert die Notwendigkeit weiterer Forschung, um mehr nutzbringendes Wissen über Code Smells zu gewinnen sowie besserer Identifikationstools, welche eine möglichst niedrigschwellige Einsatzhürde besitzen sollten.

In allen vier untersuchten Kategorien konnte die jeweilige Verbreitung der einzelnen

untersuchten Smells analysiert werden. Kategorieübergreifend betrachtet sind *Properties with Hard-Coded Values*, *High Specificity Values*, *Duplicated Code*, *Usage of !important*, *Excessive Global Variables*, *Depth*, *Lengthy Lines*, *Too Long Rules*, *Argument Count Mismatch* und *Mysterious Name* die zehn häufigsten der untersuchten Smells im Web.

Hinsichtlich der Verbreitung von Code Smells wurde die Gültigkeit des *Paretoprinzips* nachgewiesen. Somit zeigt sich, dass die Priorisierung weniger, aber sehr häufiger Smells in Entwickler-Tools und der Lehre bereits massive Effekte auf die gesamte Verbreitung von Code Smells im Web haben könnte. Die Ergebnisse dieser Arbeit zeigen, welche Smells sich für eine solche Priorisierung am besten eignen würden.

Auch bei der Analyse verschiedener Smell-Eigenschaften und ihrer jeweiligen Verbreitung konnten Ergebnisse gewonnen werden, welche in zukünftigen Arbeiten und Tools sowie der Lehre Anwendung finden können. So ergab sich die Tendenz, dass – im Gesamtkontext der in dieser Arbeit getätigten Analyse – Code Smells, die unnötigen Code thematisieren oder einen fehlerähnlichen Charakter aufweisen, häufiger verbreitet sind als Smells, die den Code optisch verschlechtern, inhaltlich schlechte Lösungen darstellen oder die Nutzung veralteter Elemente thematisieren. Hinsichtlich der extrem stark verbreiteten CSS-Smells wurde deutlich, dass diese noch viel häufiger in Deklarationen als in Selektoren auftreten.

Mit den präsentierten Ergebnissen konnten alle Ziele dieser Arbeit in einer sehr zufriedenstellenden Art und Weise erreicht werden. Dennoch sollte stets bedacht werden, in welchem Entscheidungsrahmen und Experimentumfeld diese konkreten Ergebnisse erzielt wurden. Für eine Bestätigung der Erkenntnisse bieten sich daher weitere Untersuchungen in einem alternativen Umfeld – etwa durch die Nutzung eines anderen Identifikationstools – an.

Aus den in Kapitel 7.3.5 präsentierten Ergebnissen resultierte die Hypothese, dass Smells mit einer höheren Bekanntheit weniger stark verbreitet sind. Da diese These lediglich auf einem subjektiven Eindruck basiert, bietet es sich an, diesbezüglich weitere Nachforschungen anzustellen. So könnte die Bekanntheit verschiedener Smells in Interviews mit Entwickler*innen quantifiziert werden und mit der analysierten Verbreitung der Smells abgeglichen werden.

Es ist stets zu bedenken, dass sich die Bedeutung von Code Smells nicht allein aus der hier analysierten Verbreitung ergibt. Um eine insgesamt sinnvolle Priorisierung von Code Smells vornehmen zu können, spielen weitere Faktoren eine entscheidende Rolle. Die Aspekte der Schädlichkeit oder Überlebensdauer verschiedener Code Smells wurden zum Beispiel bereits in kleinerem Umfang in anderen Arbeiten analysiert. Als Thema zukünftiger Forschung empfiehlt sich die Kombination der Ergebnisse solcher Arbeiten mit denen *dieser* Arbeit, um die Relevanz einzelner Code Smells ganzheitlich zu betrachten. Den Besonderheiten der Web-Entwicklung sollte man in weiteren Untersuchungen dieses Themengebiets gerecht werden.

Literaturverzeichnis

[AC20] AGRAHARI, Vartika ; CHIMALAKONDA, Sridhar: An Exploratory Study of Code Smells in Web Games. In: *arXiv preprint arXiv:2002.05760* (2020)

[AKGA11] ABBES, Marwen ; KHOMH, Foutse ; GUEHENEUC, Yann-Gael ; ANTONIOL, Giuliano: An empirical study of the impact of two antipatterns, blob and spaghetti code, on program comprehension. In: *2011 15Th european conference on software maintenance and reengineering* IEEE, 2011, S. 181–190

[AL20] ALMASHFI, Nabil ; LU, Lunjin: Code smell detection tool for Java Script programs. In: *2020 5th International Conference on Computer and Communication Systems (ICCCS)* IEEE, 2020, S. 172–176

[Ble18] BLEISCH, Tobias P.: An empirical study of CSS code smells in web frameworks. (2018)

[BOM20] BESSGHAIER, Narjes ; OUNI, Ali ; MKAOUER, Mohamed W.: On the diffusion and impact of code smells in web applications. In: *International Conference on Services Computing* Springer, 2020, S. 67–84

[BQO+12] BAVOTA, Gabriele ; QUSEF, Abdallah ; OLIVETO, Rocco ; DE LUCIA, Andrea ; BINKLEY, David: An empirical analysis of the distribution of unit test smells and their impact on software maintenance. In: *2012 28th IEEE International Conference on Software Maintenance (ICSM)* IEEE, 2012, S. 56–65

[BR16] BAVOTA, Gabriele ; RUSSO, Barbara: A large-scale empirical study on self-admitted technical debt. In: *Proceedings of the 13th international conference on mining software repositories*, 2016, S. 315–326

[CCMX16] CHEN, Zhifei ; CHEN, Lin ; MA, Wanwangying ; XU, Baowen: Detecting code smells in Python programs. In: *2016 international conference on Software Analysis, Testing and Evolution (SATE)* IEEE, 2016, S. 18–23

[CM10] CHATZIGEORGIOU, Alexander ; MANAKOS, Anastasios: Investigating the evolution of bad smells in object-oriented code. In: *2010 Seventh International Conference on the Quality of Information and Communications Technology* IEEE, 2010, S. 106–115

[Cod] CODEHAWK: *Codehawk-CLI*. `https://github.com/sgb-io/codehawk-cli`. – Zuletzt abgerufen am 6. Mai 2022

[Col19] COLLINS, Victoria: *The decline of the native app and the rise of the web app*. https://www.forbes.com/sites/victoriacollins/2019/04/05/why-you-dont-need-to-make-an-app-a-guide-for-startups-who-want-to-make-an-app/?sh=3cf2182d6e63. Version: Apr 2019. – Zuletzt abgerufen am 18. Februar 2022

[Cro] CROCKFORD, Douglas: *The JavaScript code quality and coverage tool*. https://www.jslint.com/. – Zuletzt abgerufen am 6. Mai 2022

[DST14] DUNFORD, Rosie ; SU, Quanrong ; TAMANG, Ekraj: The pareto principle. (2014)

[ESL] ESLINT: *Pluggable javascript linter*. https://eslint.org/. – Zuletzt abgerufen am 6. Mai 2022

[Fac] FACEBOOK: *FLOW IS A STATIC TYPE CHECKER FOR JAVASCRIPT.* https://flow.org/. – Zuletzt abgerufen am 6. Mai 2022

[FM13] FARD, Amin M. ; MESBAH, Ali: Jsnose: Detecting javascript code smells. In: *2013 IEEE 13th international working conference on Source Code Analysis and Manipulation (SCAM)* IEEE, 2013, S. 116–125

[Fow20] FOWLER, Martin: *Refactoring: Wie Sie das Design bestehender Software verbessern*. mitp Verlags GmbH & Co. KG, 2020

[Gha14] GHARACHORLU, Golnaz: *Code Smells in Cascading Style Sheets: An Empirical Study and a Predictive Model*, The University of Tehran, Diss., 2014

[GJ09] GUPTA, Pooja ; JOHARI, Kalpana: Implementation of web crawler. In: *2009 Second International Conference on Emerging Trends in Engineering & Technology* IEEE, 2009, S. 838–843

[Har12] HAROLD, Elliotte R.: *Refactoring html: improving the design of existing web applications*. Addison-Wesley Professional, 2012

[HLMHC17] HSIEH, Chin-Yun ; LE MY, Canh ; HO, Kim T. ; CHENG, Yu C.: Identification and refactoring of exception handling code smells in JavaScript. In: *Journal of Internet Technology* 18 (2017), Nr. 6, S. 1461–1471

[Inta] INTERNETLIVESTATS.COM: *Internet users*. https://www.internetlivestats.com/internet-users/. – Zuletzt abgerufen am 18. Februar 2022

[Intb] INTERNETLIVESTATS.COM: *Total number of websites*. https://www.internetlivestats.com/total-number-of-websites/#trend. – Zuletzt abgerufen am 18. Februar 2022

[JGHK13] JAAFAR, Fehmi ; GUÉHÉNEUC, Yann-Gaël ; HAMEL, Sylvie ; KHOMH, Foutse: Mining the relationship between anti-patterns dependencies and fault-proneness. In: *2013 20th working conference on reverse engineering (WCRE)* IEEE, 2013, S. 351–360

[JKA19] JOHANNES, David ; KHOMH, Foutse ; ANTONIOL, Giuliano: A large-scale empirical study of code smells in JavaScript projects. In: *Software Quality Journal* 27 (2019), Mar, 1271–1314. `https://link.springer.com/article/10.1007/s11219-019-09442-9`

[KA18] KINNE, Jan ; AXENBECK, Janna: Web mining of firm websites: A framework for web scraping and a pilot study for Germany. In: *ZEW-Centre for European Economic Research Discussion Paper* (2018), Nr. 18-033

[KDPG09] KHOMH, Foutse ; DI PENTA, Massimiliano ; GUEHENEUC, Yann-Gael: An exploratory study of the impact of code smells on software change-proneness. In: *2009 16th Working Conference on Reverse Engineering* IEEE, 2009, S. 75–84

[Kov] KOVALYOV, Anton: *JSHint, a JavaScript code quality tool*. `https://jshint.com/`. – Zuletzt abgerufen am 6. Mai 2022

[KPGA12] KHOMH, Foutse ; PENTA, Massimiliano D. ; GUÉHÉNEUC, Yann-Gaël ; ANTONIOL, Giuliano: An exploratory study of the impact of antipatterns on class change-and fault-proneness. In: *Empirical Software Engineering* 17 (2012), Nr. 3, S. 243–275

[Kro] KRONUZ: *esprima-python*. `https://github.com/Kronuz/esprima-python`. – Zuletzt abgerufen am 31. Mai 2022

[LR15] LAVALLÉE, Mathieu ; ROBILLARD, Pierre N.: Why good developers write bad code: An observational case study of the impacts of organizational factors on software quality. In: *2015 IEEE/ACM 37th IEEE International Conference on Software Engineering* Bd. 1 IEEE, 2015, S. 677–687

[Mar09] MARTIN, Robert C.: *Clean code: a handbook of agile software craftsmanship*. Pearson Education, 2009

[Mar19] MARCUS, Scott: *What is the difference between HTML event attributes and assign events using the HTML dom?* `https://stackoverflow.com/a/54497835`. Version: Feb 2019. – Zuletzt abgerufen am 1. April 2022

[MBC14] MARTINI, Antonio ; BOSCH, Jan ; CHAUDRON, Michel: Architecture technical debt: Understanding causes and a qualitative model. In: *2014 40th EUROMICRO Conference on Software Engineering and Advanced Applications* IEEE, 2014, S. 85–92

[Moh20] MOHAN, Harish: *The rise of web apps: App development company: Perfomatix: Product Engineering Services Company*. https://www.perfomatix.com/the-rise-of-web-apps-app-development-company/. Version: Sep 2020. – Zuletzt abgerufen am 18. Februar 2022

[NNN+12] NGUYEN, Hung V. ; NGUYEN, Hoan A. ; NGUYEN, Tung T. ; NGUYEN, Anh T. ; NGUYEN, Tien N.: Detection of embedded code smells in dynamic web applications. In: *2012 Proceedings of the 27th IEEE/ACM International Conference on Automated Software Engineering* IEEE, 2012, S. 282–285

[OCAD21] OORT, Bart van ; CRUZ, Luís ; ANICHE, Maurício ; DEURSEN, Arie van: The Prevalence of Code Smells in Machine Learning projects. In: *2021 IEEE/ACM 1st Workshop on AI Engineering-Software Engineering for AI (WAIN)* IEEE, 2021, S. 1–8

[OCBZ09] OLBRICH, Steffen ; CRUZES, Daniela S. ; BASILI, Victor ; ZAZWORKA, Nico: The evolution and impact of code smells: A case study of two open source systems. In: *2009 3rd international symposium on empirical software engineering and measurement* IEEE, 2009, S. 390–400

[PAN+19] PERUMA, Anthony ; ALMALKI, Khalid S. ; NEWMAN, Christian D. ; MKAOUER, Mohamed W. ; OUNI, Ali ; PALOMBA, Fabio: On the distribution of test smells in open source android applications: An exploratory study. (2019)

[PBP+18] PALOMBA, Fabio ; BAVOTA, Gabriele ; PENTA, Massimiliano D. ; FASANO, Fausto ; OLIVETO, Rocco ; LUCIA, Andrea D.: On the diffuseness and the impact on maintainability of code smells: a large scale empirical investigation. In: *Empirical Software Engineering* 23 (2018), Nr. 3, S. 1188–1221

[PVZ16] PUNT, Leonard ; VISSCHER, Sjoerd ; ZAYTSEV, Vadim: The A? B* A pattern: Undoing style in CSS and refactoring opportunities it presents. In: *2016 IEEE International Conference on Software Maintenance and Evolution (ICSME)* IEEE, 2016, S. 67–77

[PZ12] PETERS, Ralph ; ZAIDMAN, Andy: Evaluating the lifespan of code smells using software repository mining. In: *2012 16th European conference on software maintenance and reengineering* IEEE, 2012, S. 411–416

[RA15] RASOOL, Ghulam ; ARSHAD, Zeeshan: A review of code smell mining techniques. In: *Journal of Software: Evolution and Process* 27 (2015), Nr. 11, S. 867–895

[RA17] RASOOL, Ghulam ; ARSHAD, Zeeshan: A lightweight approach for detection of code smells. In: *Arabian Journal for Science and Engineering* 42 (2017), Nr. 2, S. 483–506

[Ric] RICHARDSON, Leonard: *Beautiful Soup documentation*. `https://www.crummy.com/software/BeautifulSoup/bs4/doc/`. – Zuletzt abgerufen am 30. Mai 2022

[Rob12] ROBBINS, Jennifer N.: *Learning web design: A beginner's guide to HTML, CSS, JavaScript, and web graphics*. Ö'Reilly Media, Inc.", 2012

[RR22] RACHOW, Paula ; RIEBISCH, Matthias: An architecture smell knowledge base for managing architecture technical debt. In: *Proceedings of the International Conference on Technical Debt*, 2022, S. 1–10

[Sab16] SABOURY, Amir: *On the Fault-Proneness of Javascript Code Smells*, Ecole Polytechnique, Montreal (Canada), Diss., 2016

[Sap] SAPIN, Simon: *tinycss2*. `https://doc.courtbouillon.org/tinycss2/stable/index.html`. – Zuletzt abgerufen am 16. Juni 2022

[Sar15] SAREEN, Himanshu: *How apps overthrew web development and changed the internet*. `https://www.wired.com/insights/2014/06/apps-overthrew-web-development-changed-internet/`. Version: Aug 2015. – Zuletzt abgerufen am 18. Februar 2022

[Sch] SCHRÖDER, Janik: *WebDeo*. `https://git.informatik.uni-hamburg.de/6schroed/code-smell-detection-web-development`. – Zuletzt abgerufen am 3. Oktober 2022

[Sch21] SCHRÖDER, Janik: *Analyse der Nutzung von veraltetem HTML-Code im World Wide Web*. 2021

[Shv19] SHVETS, Alexander: *Dive Into Refactoring*. Bd. V2019-1.3. Refactoring.Guru, 2019

[SL06] SHATNAWI, Raed ; LI, Wei: An investigation of bad smells in object-oriented design. In: *Third International Conference on Information Technology: New Generations (ITNG'06)* IEEE, 2006, S. 161–165

[SMK17] SHOENBERGER, Ian ; MKAOUER, Mohamed W. ; KESSENTINI, Marouane: On the use of smelly examples to detect code smells in JavaScript. In: *European Conference on the Applications of Evolutionary Computation* Springer, 2017, S. 20–34

[SMKA17] SABOURY, Amir ; MUSAVI, Pooya ; KHOMH, Foutse ; ANTONIOL, Giulio: An empirical study of code smells in javascript projects. In: *2017 IEEE 24th international conference on software analysis, evolution and reengineering (SANER)* IEEE, 2017, S. 294–305

[Sona] SONARQUBE: *Code Quality and Code Security*. https://www.sonarqube.org/. – Zuletzt abgerufen am 6. Mai 2022

[Sonb] SONARSOURCE: *Control Flow Statements ïf", for", "while", ßwitchänd tryß-hould not be nested too deeply*. https://rules.sonarsource.com/javascript/RSPEC-134. – Zuletzt abgerufen am 23. März 2022

[SS18] SHARMA, Tushar ; SPINELLIS, Diomidis: A Survey on Software Smells. In: *Journal of Systems and Software* (2018), Nov

[SSS14] SURYANARAYANA, Girish ; SAMARTHYAM, Ganesh ; SHARMA, Tushar: *Refactoring for software design smells: managing technical debt*. Morgan Kaufmann, 2014

[SV21] SUGANYA, E ; VIJAYARANI, S: Firefly Optimization Algorithm Based Web Scraping for Web Citation Extraction. In: *Wireless Personal Communications* 118 (2021), Nr. 2, S. 1481–1505

[TAV13] TOM, Edith ; AURUM, Aybüke ; VIDGEN, Richard: An exploration of technical debt. In: *Journal of Systems and Software* 86 (2013), Nr. 6, S. 1498–1516

[Tea] TEAM, Esprima D.: *Esprima*. https://docs.esprima.org/en/latest/. – Zuletzt abgerufen am 31. Mai 2022

[TN11] TITTEL, Ed ; NOBLE, Jeff: *HTML, XHTML and CSS for dummies*. John Wiley & Sons, 2011

[TPB+15] TUFANO, Michele ; PALOMBA, Fabio ; BAVOTA, Gabriele ; OLIVETO, Rocco ; DI PENTA, Massimiliano ; DE LUCIA, Andrea ; POSHYVANYK, Denys: When and why your code starts to smell bad. In: *2015 IEEE/ACM 37th IEEE International Conference on Software Engineering* Bd. 1 IEEE, 2015, S. 403–414

[w3sa] W3SCHOOLS: *AJAX Introduction*. https://www.w3schools.com/js/js_ajax_intro.asp. – Zuletzt abgerufen am 9. März 2022

[w3sb] W3SCHOOLS: *CSS Selector Reference*. https://www.w3schools.com/cssref/css_selectors.asp. – Zuletzt abgerufen am 31. März 2022

[w3sc] W3SCHOOLS: *CSS The !important Rule*. https://www.w3schools.com/css/css_important.asp. – Zuletzt abgerufen am 31. März 2022

[w3sd] W3SCHOOLS: *HTML Event Attributes*. https://www.w3schools.com/tags/ref_eventattributes.asp. – Zuletzt abgerufen am 31. Mai 2022

[w3se] W3SCHOOLS: *HTML <h1> to <h6> Tags*. https://www.w3schools.com/tags/tag_hn.asp. – Zuletzt abgerufen am 11. April 2022

[w3sf] W3SCHOOLS: *JavaScript Scope*. `https://www.w3schools.com/js/js_scope.asp`. – Zuletzt abgerufen am 17. Mai 2022

[wha] WHATSMYIP.ORG: *Random Website Machine*. `https://www.whatsmyip.org/random-website-machine/`. – Zuletzt abgerufen am 13. Juni 2022

[Yin18] YIN, Robert K.: *Case study research and applications - Design and methods*. Bd. 6. sage, 2018

[YM12] YAMASHITA, Aiko ; MOONEN, Leon: Do code smells reflect important maintainability aspects? In: *2012 28th IEEE international conference on software maintenance (ICSM)* IEEE, 2012, S. 306–315

[YM13] YAMASHITA, Aiko ; MOONEN, Leon: Do developers care about code smells? An exploratory survey. In: *2013 20th working conference on reverse engineering (WCRE)* IEEE, 2013, S. 242–251

Abbildungsverzeichnis

Tabellenverzeichnis

Abkürzungsverzeichnis

AJAX Asynchronous JavaScript And XML

AST Abstract Syntax Tree

CBD Callback Depth

CSS Cascading Style Sheets

CpF Characters per File

CpL Characters per Line

CpS Characters per Smell

CpW Characters per Website

CV Variationskoeffizient

DDoS Distributed Denial of Service

DOM Document Object Model

FpW Files per Website

HTML Hypertext Markup Language

ID Identifier

ISI Identified Smell Instances

JS JavaScript

LOC Lines of Code

LpF Lines per File

LpS Lines per Smell

LpW Lines per Website

MAX Maximalwert

MIN Minimalwert

NCM Number of Chained Methods

NCLS Number of Class Selectors

NCS Number of Case Statements

NES Number of Element Selectors

NGV Number of Global Variables

NIS Number of ID Selectors

NNB Number of Nested Blocks

NoAS Number of Analyzed Smells

NOC Number of Characters

NOD Number of Declarations

NOP Number of Parameters

NOSU Number of Selector Units

NOV Number of Values

RQ Forschungsfrage

SmeII Code Smell

SoC Separation of Concerns

SpF Smells per File

SpW Smells per Website

SQL Structured Query Language

$\bar{x}$ Mittelwert

$\tilde{x}$ Median

URL Uniform Resource Locator

A. Anhang

A.1. Übersicht der Eigenschaften von Code Smells

Code Smell	CSS *	Deklaration	Fehlerähnlich	HTML *	Inhaltlich	JavaScript *	Optisch	Selektor	SoC	Überflüssig	Veraltet
Argument Count Mismatch			✓			✓					
Array Length Assignment			✓			✓					
Assignment in Conditional Statement			✓			✓					
Chained Methods					✓	✓	✓				
Commented-Out Code	✓			✓		✓	✓			✓	
Complex Switch Case						✓	✓				
CSS in HTML Style Attribute	✓			✓					✓		
CSS in HTML Style Tag	✓			✓					✓		
CSS in JavaScript	✓					✓			✓		
Deprecated Attributes				✓							✓
Deprecated Styling Tags	✓			✓					✓		✓
Deprecated Tags				✓							✓
Depth						✓	✓				
Duplicated Code	✓					✓	✓			✓	
Empty Catch						✓				✓	
Empty Rules	✓						✓			✓	
Excessive Global Variables			✓			✓					
High Specificity Values	✓							✓			
HTML in JavaScript				✓		✓	✓		✓		
JavaScript in HTML Events				✓		✓			✓		
JavaScript in HTML Links				✓		✓			✓		
JavaScript in HTML Script Tag				✓		✓			✓		
Lengthy Lines	✓	✓		✓		✓	✓				
Long Method						✓	✓				
Long Parameter List					✓	✓					
Mysterious Name				✓		✓					
Nested Callbacks					✓	✓	✓				
Opening Tag missing Closing Tag			✓	✓			✓				
Properties with Hard-Coded Values	✓	✓									
Repeated Switches						✓	✓			✓	
Selectors with ID and Class or Element	✓				✓			✓		✓	
Skipped Headings				✓							
This Assign						✓					
Too general Selectors	✓							✓			
Too long Rules	✓						✓				
Too much Cascading	✓							✓			
Universal Selectors	✓							✓			
Uppercase Tags				✓							
Usage of !important	✓	✓									
Insgesamt	16	3	5	15	4	22	13	5	8	6	3

Tabelle A.1.: Übersicht der untersuchten Code Smells und ihrer Eigenschaften

* Die genaue Bedeutung der Labels ist zu beachten (⇒ Kapitel 5.1). Sprach-Labels sind aufgrund der Separation-Smells nicht gleichbedeutend mit der Anzahl untersuchter Smells je Sprache.

A.2. Zusätzliche Informationen zu ausgewählten Code Smells

Im diesem Abschnitt werden zusätzliche Informationen zu einigen Code Smells in alphabetischer Reihenfolge präsentiert. Dabei handelt es sich meistens um Beispiele, die beim Verständnis jener Code Smells helfen sollen, welche durch die Beschreibung in reiner Textform, schwierig in Gänze zu erfassen sein könnten. Außerdem werden für einige Smells zusätzliche Daten zur Identifikation benötigt. Diese werden als Liste mit zusätzlicher Referenz ihres Ursprungs angegeben.

Deprecated Attributes

Liste aller generell veralteten Attribute [Sch21]:
`abbr, align, alink, archive, axis, background, bgcolor, border, cellpadding, cellspacing, char, charoff, clear, code, codebase, codetype, compact, declare, frame, frameborder, hspace, language, link, longdesc, marginheight, marginwidth, nohref, noshade, nowrap, profile, rev, rules, scheme, scrolling, standby, text, valign, valuetype, version, vlink, vspace`

Liste veralteter Attribut-Tag-Kombinationen [Sch21]:
`<a coords>, <a shape>, <a name>, <image name>, <a charset>, <a link>, <col width>, <hr width>, <pre width>, <table width>, <td width>, <th width>, <hr size>, <input usemap>, <li type>, <ol type>, <ul type>, <param type>, <link target>, <td height>, <th height>`

Deprecated Styling Tags

Liste aller veralteten Tags mit Styling-Bezug [Sch21]:
`basefont, big, blink, center, font, strike, tt`

Deprecated Tags

Liste aller veralteten Tags ohne Styling-Bezug [Sch21]:
`acronym, applet, dir, fn, frame, frameset, isindex, listing, menu, menuitem, noframes, xmp`

High Specificity Values

Beispiel $NIS > 1$: `#test #test2 { ... }`
Beispiel $NCLS > 2$: `.test .test2 .test3 { ... }`
Beispiel $NES > 3$: `body div p span { ... }`

HTML in JavaScript

```
var s = document.createElement( "script" );
s.type = "text/javascript";
s.src = url;
document.body.appendChild( s );
```

Abbildung A.1.: Beispiel: HTML in JavaScript durch die Nutzung von `createElement()`

```
var defaultTmplLoggedIn = "<div class='mrp-listings-user-panel'>" +
    "<label>Logged in as:</label>" +
    "<span class='logged-in-username'></span> "+
    "<a class='logged-in-account-link' href='javascript:;'>Account</a> "+
    "<a class='logged-in-logout-link' href='javascript:;'>Logout</a>"+
    "</div>";
```

Abbildung A.2.: Beispiel: HTML in JavaScript durch HTML-Code als String

JavaScript in HTML Events

Beispiel:

```
<input id='toggleAcronyms' onclick='toggleAcronyms();' type='checkbox'/>
```

Liste aller 71 HTML-Event-Attribute: [w3sd]

JavaScript in HTML Links

Beispiel nach [NNN+12]:

```
<a href='javascript:history.back()'>
```

Mysterious Name

Limitationen:
Statische Code-Analyse ermöglicht keine Bezeichnerinterpretation, lediglich eine Analyse der Länge der Bezeichner. HTML-IDs, -Klassen sowie JavaScript-Methoden und -Klassen sollten nie nur nach Buchstaben benannt werden. Für Variablen kann dies sinnvoll sein (Iteratoren, mathematische Formeln, ...).

Nested Callbacks

```
setTimeout(function () {
    xhr("/greeting/", function (greeting)
    {
        xhr("/who/?greeting=" + greeting, function (who)
        {
            xhr("/when/?greeting=" + who, function (when)
            {
                document.write(greeting + " " + who + " " + when);
            });
        });
    });
}, 1000);
```

Abbildung A.3.: Beispiel: Nested Callbacks (nach [FM13])

Selectors with ID and Class or Element

Beispiele:

`.text #bold { ... }, span#bold { ... }, span #bold { ... }, .text#bold { ... }`

This Assign

```
function user(id) {
    var self = this; // This assign
    self.id = id;
    getPropertiesById (id , function (props) {
        self.props = props;
    });
}
```

Abbildung A.4.: Beispiel: This Assign

Too general Selectors

Liste aller Selektoren, die als zu allgemein gelten [Gha14]:

`html, head, body, div, header, aside`

Eidesstattliche Versicherung

Hiermit versichere ich an Eides statt, dass ich die vorliegende Arbeit im Masterstudiengang Wirtschaftsinformatik selbstständig verfasst und keine anderen als die angegebenen Hilfsmittel – insbesondere keine im Quellenverzeichnis nicht benannten Internet-Quellen – benutzt habe. Alle Stellen, die wörtlich oder sinngemäß aus Veröffentlichungen entnommen wurden, sind als solche kenntlich gemacht. Ich versichere weiterhin, dass ich die Arbeit vorher nicht in einem anderen Prüfungsverfahren eingereicht habe und die eingereichte schriftliche Fassung der elektronischen Abgabe entspricht.
Ich stimme der Einstellung der Arbeit in die Bibliothek des Fachbereichs Informatik zu.

Ebstorf, den 24.10.2022 Unterschrift: _____