

INHALTSVERZEICHNIS

I Thesis	
1 Motivation	2
2 Grundlagen	3
2.1 Software-Qualitätsmetriken	3
2.2 Software-Visualisierung	5
2.2.1 3D-Software-Visualisierung	5
2.3 CodeCity: Grundstein für 3D-Visualisierung	7
2.4 Treemap-Layouts	8
2.4.1 Squarify-Algorithmus	10
3 Problemstellung	19
3.1 Evaluationsrahmen	19
3.2 Das Treemap-Problem	21
4 Verwandte Arbeiten	25
4.0.1 Vergleich von Visualisierungstechniken	25
4.1 Treemap Problem verwandte Arbeiten	29
5 Layout Suche	33
5.1 Methodik	33
5.2 Durchführung der Suche	35
5.3 Dokumentation der Ergebnisse und Vorstellung der Layout-Umsetzung	46
5.3.1 Sunburst Layout	46
5.3.2 Circle Packing Layout	46
5.3.3 Noch eins mehr?	46
6 Datenanalyse	47
6.1 Auswahl der Projekte	47
6.1.1 Die Auswahl der Projekte im Detail	51
6.2 Struktur von Softwareprojekten	53
6.2.1 Orderketten	54
7 Erweiterung des Squarify Algorithmus	57
7.1 Kriterien für Treemap Layouts	57
7.2 Zweifache Berechnung	58
7.2.1 annäherungs wert	59
7.2.2 Größenanpassung	59
7.3 Scaling der Knoten	62
7.4 Reihenfolge der Knoten im ersten Layoutschritt	63
7.4.1 Reihenfolge der Knoten nach erstem Layoutschritt	65
7.5 Mehrfache Berechnung	66
7.5.1 Iterative Abstandsvergrößerung	67
7.6 Änderungen im Layout	67
7.6.1 Optimierung von Ordnerketten	67
7.6.2 Beschriftung der Knoten	68

7.6.3 Abstände zwischen Geschwister Knoten	72
7.7 Fazit	73
8 evaluation	77
9 Fazit	78
II Appendix	
A Dateien	81
Literatur	96

Teil I
THESIS

MOTIVATION

Die Qualität von Software für Menschen verständlich und greifbar zu machen, die nicht täglich mit Quellcode arbeiten, stellt nach wie vor eine Herausforderung dar. Aber auch erfahrene Entwickler*innen stehen regelmäßig vor der Aufgabe, sich schnell einen Überblick über ein bestehendes Software-System zu verschaffen: Wo liegen die problematischen Stellen? Wo lohnt sich ein tieferer Blick? Und wie lassen sich Verbesserungspotenziale effizient identifizieren?

Um solche Fragen beantworten zu können, sind geeignete Visualisierungen essenziell. Sie helfen, komplexe Strukturen zu abstrahieren, Muster zu erkennen und technische Schulden sichtbar zu machen, ohne dass zunächst jede Zeile Code gelesen werden muss. Während einfache Diagramme oder Dashboards nützliche Einstiegspunkte bieten, zeigen immersive Ansätze, insbesondere dreidimensionale Darstellungen, ein deutlich höheres Potenzial, Software auf eine intuitivere und erfahrbare Weise zugänglich zu machen [MFM03; WLo7; Rei95; KMoo].

Insbesondere 3D-Metaphern, wie die Stadt-Metapher [WLo7], haben sich in Forschung und Praxis als äußerst wirkungsvoll erwiesen (QUELLEN). Durch die Übertragung von Softwarestrukturen auf städtische Elemente (Gebäude, Blöcke, Straßen) entsteht ein räumliches Abbild, das verschiedene Code-Metriken kombinieren kann. Die Beliebtheit dieser Methode beruht nicht zuletzt auf der hohen Informationsdichte und der intuitiven Lesbarkeit räumlicher Strukturen. Im Vergleich dazu wirken alternative Metaphern, etwa waldartige Darstellungen [Atz+21], häufig unübersichtlich und verlieren bei wachsender Komplexität schnell an Aussagekraft.

Gleichzeitig zeigt sich, dass auch die Stadt-Metapher nicht frei von Schwächen ist. Gerade bei sehr großen Codebasen stößt die Übersichtlichkeit an ihre Grenzen oder wichtige Details gehen verloren [LFo8]. Viele dieser Visualisierungen beruhen auf Treemap-Layouts, einer etablierten Technik zur Darstellung hierarchischer Strukturen. Doch gerade in Bezug auf Lesbarkeit, Informationsdichte und Benutzerfreundlichkeit stellt sich zunehmend die Frage, ob der klassische Treemap-Ansatz nicht in Bezug auf 3D-Softwarevisualisierungen verbessert werden kann.

Außerdem ist offen, ob es alternative Layout-Ansätze gibt, die eine noch klarere, verständlichere oder flexiblere Darstellung sowohl in 2D als auch in 3D ermöglichen würden. In der Praxis ist dieser Aspekt bisher wenig beleuchtet worden, obwohl er entscheidend für die Nützlichkeit solcher Visualisierungen ist.

GRUNDLAGEN

In diesem Abschnitt werden die grundlegenden Konzepte und Begriffe erläutert, die für das Verständnis dieser Arbeit notwendig sind. Zunächst werden Software-Qualitätsmetriken vorgestellt, da sie die Basisdaten für alle in dieser Arbeit genutzten Visualisierungen liefern. Anschließend wird das Thema Software-Visualisierung behandelt, wobei insbesondere auf die dreidimensionale Software-Visualisierung mit Hilfe von Treemaps eingegangen wird.

2.1 *Software-Qualitätsmetriken*

Software-Qualitätsmetriken sind zentrale Werkzeuge zur Messung und Bewertung der Qualität von Software. Sie ermöglichen es, verschiedene Eigenschaften der Software objektiv anhand von Kennzahlen zu analysieren. Eine *Softwaremetrik* ist dabei meist als mathematische Funktion zu verstehen, die eine spezifische Eigenschaft einer Software in einen Zahlenwert abbildet:

Eine Softwaremetrik, oder kurz Metrik, ist eine (meist mathematische) Funktion, die eine Eigenschaft von Software in einen Zahlenwert, auch Maßzahl genannt, abbildet. Hierdurch werden formale Vergleichs- und Bewertungsmöglichkeiten geschaffen.[WP04]

Auch in anderen Definitionen wird die grundsätzliche Funktion von Metriken hervorgehoben:

Softwaremetrik ist ein quantitatives Maß, das verwendet wird, um die Eigenschaften eines Softwareproduktes oder des Softwareentwicklungsprozesses zu bewerten.[Sofa]

Eine Softwarequalitätsmetrik ist eine Funktion, die eine Software-Einheit in einen Zahlenwert abbildet, welcher als Erfüllungsgrad einer Qualitätseigenschaft der Software-Einheit interpretierbar ist.[EH98]

Allen Definitionen ist gemeinsam, dass eine Metrik stets nur einen bestimmten Aspekt der Software abbildet. Genau darin liegt eine der wichtigsten Erkenntnisse und gleichzeitig eine zentrale Kritik: Einzelne Metriken geben ausschließlich Auskunft über spezifische Eigenschaften und erlauben keine umfassende Bewertung der gesamten Softwarequalität.

Zusammenfassend kann festgestellt werden, dass Software-Qualitätsmetriken als quantitative Indikatoren zur Kontrolle und Verbesserung der Software dienen. Sie sind jedoch stets darauf beschränkt, nur bestimmte Aspekte zu messen. Die Aussagekraft jeder einzelnen Metrik ist daher begrenzt:

Eine Metrik alleine kann nie eine vollständige Aussage über die Qualität der Software treffen. Metriken sollten immer in Kombination bewertet werden. Die Güte und Reife von Software kann nur in Kombination mit statischer Code Analyse, Code Reviews und funktionalen Tests final beurteilt werden.[Sch19]

Es existiert also keine universelle Metrik, die volumnfänglich Auskunft über die Qualität einer Software geben kann. Dies allein schon, da der Begriff *gute Software* situationsabhängig definiert werden muss. Selbst wenn fehlerfreie Ausführung als Ziel gilt, ist unklar, wie diese Eigenschaft in eine einzelne Metrik gefasst werden könnte.

Um ein möglichst vollständiges Qualitätsbild zu erhalten, ist es daher essenziell, verschiedene Metriken miteinander zu kombinieren. Die parallele Interpretation mehrerer Indikatoren erhöht die Aussagekraft und reduziert das Risiko von Fehleinschätzungen.

Die Aussagekraft von Softwaremetriken hängt maßgeblich davon ab, dass die Ziele der Software klar definiert sind [HF94]. Metriken müssen stets so gewählt werden, dass sie die Qualität einer Software bezüglich dieser Ziele abbilden. Beispielsweise ist eine Komplexitätsmetrik irrelevant, wenn primär die Ausführungsgeschwindigkeit optimiert werden soll. Auch kann die Erhebung und Auswertung von Metriken selbst mit Aufwand verbunden sein.

Ein weiterer Kritikpunkt ist, dass viele Metriken nicht ausreichend wissenschaftlich fundiert sind und der tatsächliche Nutzen nicht immer klar belegt werden kann [VK17]. Die Orientierung an Metriken sollte daher immer kritisch erfolgen und nicht zum Selbstzweck werden.

Zudem besteht die Gefahr, dass bei unpassend gewählten Metriken lediglich diese Werte optimiert werden, ohne dass sich dadurch die Gesamtqualität der Software verbessert. Entwickler könnten dazu verleitet werden, ihr Handeln auf das Erreichen guter Metrikwerte anstatt auf das eigentliche Ziel der Software auszurichten [Wor].

Nicht zuletzt sind Metriken kontextabhängig. Sie liefern Indizien, können aber Prozesse wie Code Reviews oder Tests keinesfalls ersetzen. Die Interpretation muss mit Blick auf den jeweiligen Kontext erfolgen. Bekannte Beispiele wie Lines-of-Code oder Anzahl von Commits werden oft falsch als Qualitätsindikatoren genutzt, obwohl sie weder die Komplexität noch die tatsächliche Arbeitsleistung adäquat abbilden [Wor].

Einfache Metriken können zudem keine komplexen Qualitätsprobleme identifizieren [VK17] und sind häufig stark abhängig von Programmiersprache und individuellem Programmierstil.

Die Norm ISO/IEC 25010 [Iso] beschreibt einen Rahmen, in dem die Qualität von Software anhand von acht Qualitätsmerkmalen, wie z.B. funktionaler Eignung, Zuverlässigkeit oder Benutzerfreundlichkeit, strukturiert wird. Diese Merkmale beziehen sich auf die Sicht von außen, also wie der Benutzer die Software erlebt:

Qualität wird als Abwesenheit von Fehlern im Systemverhalten verstanden, die Software verhält sich demnach so, wie der Benutzer es erwartet.[[Wit18](#), S. 1]

Für die Ursachen bestimmter Qualitätsprobleme reicht die Betrachtung aus Nutzersicht allerdings oft nicht aus. Seit den 1970er Jahren entstand daher ein Fokus auf die Messung der *internen Qualität* beziehungsweise der *Code-Qualität*, etwa mittels Metriken wie der McCabe-Komplexität [[Lud](#)]. Diese ist insbesondere für Entwickler sowie Auftraggeber relevant und unterstützt die Sicherstellung von Wartbarkeit und Erweiterbarkeit der Software. Im Rahmen dieser Arbeit wird, soweit von Metriken die Rede ist, explizit auf Code-Qualitätsmetriken Bezug genommen.

Software-Qualitätsmetriken beziehen sich in der Regel auf spezifische Softwareeinheiten, beispielsweise Dateien, Module oder Klassen. Diese Einheiten sind meistens hierarchisch, etwa entlang von Datei- und Ordnerstrukturen, organisiert. Die Messwerte werden häufig auf der Ebene einzelner Dateien gesammelt und dann auf höhere Ebenen, zum Beispiel Ordner oder Module, aggregiert, sodass eine baumartige Hierarchie entsteht.

Die im Folgenden vorgestellten Ansätze zur Visualisierung hierarchischer Metrik-Daten sind nicht zwingend spezifisch für Softwaremetriken, sondern lassen sich auch auf andere Anwendungsfälle übertragen. Dennoch weist die Struktur von Software-Einheiten und deren Metrik-Daten gewisse Eigenheiten auf, was Einfluss auf die Gestaltung der Visualisierungen haben kann (siehe Abschnitt ??).

2.2 Software-Visualisierung

Die Darstellung von Softwarestrukturen zählt zu den entscheidenden Schritten, um komplexe Softwaresysteme für Menschen verständlich zu machen. Während numerische Metriken oder tabellarische Darstellungen für Fachleute in der Softwareentwicklung oft ausreichend sind, können diese Darstellungsformen für Personen ohne technischen Hintergrund abstrakt und schwer zugänglich bleiben. Es besteht daher das Ziel, Softwarequalität auch für nicht-technische Stakeholder intuitiv und erlebbar zu machen.

HIER NOCH QUELLEN UND BISSCHEN WAS ZU DER HISTORIE UND WARUM WIESO PIPAPO

2.2.1 3D-Software-Visualisierung

Traditionell werden Softwaremetriken in numerischer oder zweidimensionaler Form präsentiert. Diese Formen liefern zwar einen guten Überblick, vermitteln jedoch wenig *Greifbarkeit* des Softwaresystems. 3D-Visualisierungen bieten dagegen das Potenzial, immersive und erfahrbare Eindrücke zu schaffen, sodass Nutzer *in die Software eintauchen* können:

Despite the proven usefulness of 2D visualizations, they do not allow the viewer to be immersed in a visualization, and the fee-

ling is that we are looking at things from 'outside'. 3D visualizations on the other hand provide the potential to create such an immersive experience [WLo7, S. 1]

Die Idee, Softwarestrukturen dreidimensional darzustellen, ist nicht neu. Bereits 1995 stellte Steven P. Reiss einen ersten Ansatz zur 3D-Visualisierung von Software vor [Rei95]. Ursprünglich zielten diese Ansätze darauf ab, Entwicklern einen schnellen Überblick über Struktur und Aufbau umfangreicher Systeme, insbesondere beim Einarbeiten in unbekannte Software, zu verschaffen [YM98]. Die frühen Methoden fokussierten dabei meist die Darstellung von Architektur und Verbindungen, weniger jedoch die Visualisierung von Softwarequalitätsmetriken (siehe Abbildung 2.1).

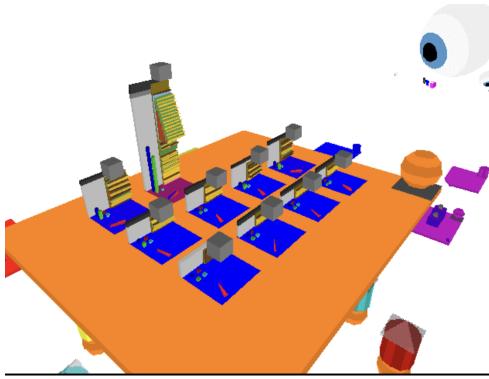


Abbildung 2.1: Beispiel für eine 3D-Visualisierung von Young und Munro [YM98, S. 6]

Für die Visualisierung von Softwarequalitätsmetriken lassen sich aus der Literatur Kriterien für eine gelungene 3D-Darstellung ableiten [YM98]:

- **Darstellung der Struktur:** Die Visualisierung soll Architektur und Aufbau der Software möglichst anschaulich und nachvollziehbar zeigen. Wichtige Aspekte dabei sind:
 - *Informationsgehalt:* Die Darstellung soll möglichst viele relevante Informationen kompakt vermitteln.
 - *Visuelle Komplexität:* Trotz hoher Informationsdichte soll die Visualisierung übersichtlich und nicht überfordernd wirken. Dies ist als Gegenspieler zum Informationsgehalt zu verstehen.
 - *Skalierbarkeit:* Auch große Softwaresysteme müssen klar und verständlich dargestellt werden können. Die Autoren von *Visualising Software in virtual reality* [YM98] sagen sogar, dass Mechanismen notwendig wären, um Komplexität und Informationsgehalt bei zunehmender Größe manuell zu steuern und je nach Software-System anpassen zu können.
 - *Stabilität gegenüber Änderungen:* Die Visualisierung soll bei Software-Änderungen konsistent bleiben, um Vergleichbarkeit über Versionen hinweg zu gewährleisten.

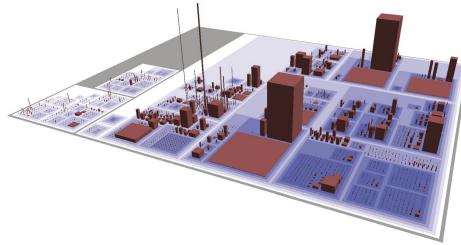


Abbildung 2.2: Beispiel für eine originale CodeCity-Visualisierung [WLo7, S. 2]

- **Visuelle Metaphern:** Der Einsatz eingängiger Metaphern erleichtert das Verständnis komplexer Softwarestrukturen.
- **Abstraktion:** Unwesentliche Details sollten ausgeblendet werden, um ein verständliches, abstrahiertes Modell zu erzeugen.
- **Navigation:** Die Orientierung innerhalb der Visualisierung muss auch bei sehr großen Systemen einfach und intuitiv möglich sein.
- **Korrelation mit dem Code:** Es sollte eine klare Zuordnung zwischen Visualisierungselementen und Quellcodebestandteilen existieren.
- **Automatisierung:** Die Generierung der Visualisierung sollte vollständig automatisierbar sein, sodass keine manuelle Nachbearbeitung notwendig ist.

2.3 CodeCity: Grundstein für 3D-Visualisierung

Eine prägende Arbeit für die 3D-Visualisierung von Softwarestrukturen und -metriken ist das Konzept *CodeCity* von Richard Wettel und Michele Lanza [WLo7]. Sie stellten erstmals einen Ansatz vor, der Software-Struktur auf Modulebene nutzt, um den Grundriss für eine 3D-Visualisierung zu schaffen. Dabei werden Softwareeinheiten als Quader dargestellt. Wie von Young und Munro [YM98] gefordert, nutzen sie eine gute visuelle Metapher, um die Software darzustellen: die Stadtmetapher. Sie ist zentrales Gestaltungsmittel von CodeCity: Klassen erscheinen als Gebäude, Pakete als Stadtbezirke. Den Objekten werden verschiedene Attribute (Dimensionen, Positionen, Farben, Farbsättigung, Transparenz) zugeordnet, was eine intuitive und greifbare Darstellung ermöglicht (siehe Abbildung 2.2).

Während frühere Arbeiten, wie die von Marcus et al. [MFM03], auf sehr feingranulare Codebestandteile, wie Attribute und Methoden, fokussierten, bietet CodeCity eine abstrahierende Sicht auf Klassen, Pakete und deren Struktur und ist dadurch vor allem für nicht-technische Stakeholder verständlicher und leichter zu überblicken.

Der Grundriss (folgend auch Layout) der Städte beruht auf einer Variante der Treemap-Algorithmen (erklärt im folgenden Abschnitt 2.4). Es ist zu erkennen, dass größere Gebäude im unteren linken Bereich, kleinere weiter

oben rechts angeordnet sind. Gebäude (Module) und Viertel (Pakete) sind immer quadratisch, dadurch bleibt viel ungenutzte *leere* Fläche.

Das Originalverfahren des CodeCity-Algorithmus ist nicht öffentlich verfügbar. In dieser Arbeit wird daher für vergleichende Analysen eine eigene Implementierung verwendet, so dass Abweichungen zur Originalvisualisierung möglich sind (außer wenn anders gekennzeichnet).

2.4 Treemap-Layouts

Hierarchische Daten werden häufig als Bäume dargestellt, jedoch zeigen sich klassische Darstellungen wie Baumdiagramme oder Venn-Diagramme insbesondere bei großen Datenmengen als ineffizient hinsichtlich der Flächennutzung und generell unübersichtlich [JS91]. Besonders für große, strukturreiche und mengenorientierte Daten stoßen traditionelle Visualisierungsmethoden an ihre Grenzen, da sie weder Informationen noch Hierarchien kompakt und intuitiv vermitteln können.

Um diese Herausforderungen zu adressieren, entwickelten Shneiderman und Johnson 1991 das Konzept der *Treemap* zur effizienten Darstellung hierarchischer Strukturen [JS91]. Das zentrale Prinzip von Treemaps besteht darin, jedem Knoten eines gewichteten Baums ein Rechteck zuzuweisen, dessen Fläche proportional zu dem Gewicht (beispielsweise Datenmenge, Marktwert, o. ä.) ist. Die Rechtecke aller Blätter füllen die Fläche des Wurzelrechtecks vollständig aus, sodass die Gesamtfläche der Kindknoten exakt der Fläche ihres Elternrechtecks entspricht. Mathematisch liegt der Visualisierung also ein gewichteter Baum zugrunde, wobei insbesondere die Blattknoten jeweils einen numerischen Wert besitzen.

Shneiderman und Johnson identifizieren für gelungene Treemaps folgende Schlüsselziele:

- **Effiziente Platznutzung:** Maximale Informationsdichte auf minimalem Raum.
- **Verständlichkeit:** Einfache Erfassbarkeit der dargestellten Hierarchie und Werte mit geringem kognitiven Aufwand. Die Struktur soll möglichst schnell erkennbar sein.
- **Ästhetik:** Ansprechende und übersichtliche Anordnung der Rechtecke.

Frühere Ansätze wie Listen, klassische Baumdiagramme (Abbildung 2.3) oder Venn-Diagramme (Abbildung 2.4) konnten diese Anforderungen nicht erfüllen. Sie leiden unter Problemen wie ineffizienter Flächenausnutzung und fehlender Möglichkeit, zusätzlich zu den Beziehungen auch quantitativermetrische Informationen anschaulich zu vermitteln. So kann zum Beispiel in klassischen Baumdiagrammen bei großen Datenmengen mehr als die Hälfte des dargestellten Bereichs aus leerem, informationslosem Raum bestehen [JS91, S. 3]. Venn-Diagramme sind insbesondere aus Platzgründen für größere Strukturen ungeeignet: "The space required between regions would cer-

tainly preclude this Venn diagram representation from serious consideration for larger structures.”[JS91, S. 5]

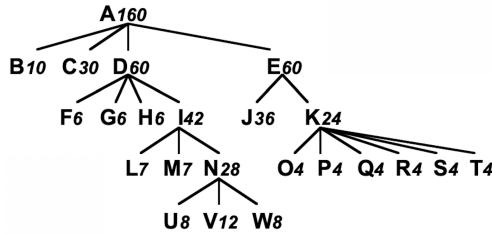


Abbildung 2.3: Beispiel für ein Baumdiagramm

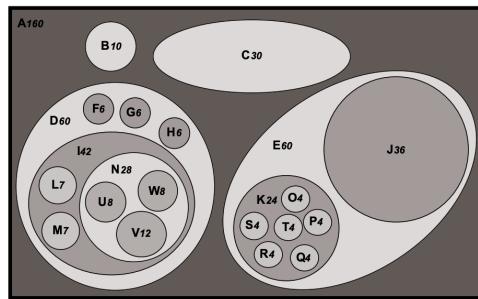


Abbildung 2.4: Beispiel für ein Venn-Diagramm

Um die genannten Schwierigkeiten zu lösen, formulieren Shneiderman und Johnson vier grundlegende Eigenschaften für Treemap-Layouts:

- Ist Knoten 1 ein Vorfahre von Knoten 2, so ist der Bereich von Knoten 2 vollständig innerhalb des Bereichs von Knoten 1 enthalten.
- Die Bereiche von Knoten schneiden sich, wenn einer ein Vorfahre des anderen ist.
- Jeder Knoten erhält eine Fläche, die streng proportional zu seinem Gewicht ist.
- Das Gewicht eines Knotens ist (mindestens) so groß wie die Summe der Gewichte seiner Kinder.

Sie präsentieren einen einfachen, rekursiven Algorithmus, der diese Bedingungen erfüllt: Der verfügbare Raum wird abwechselnd vertikal und horizontal entsprechend den Gewichten der Kindknoten aufgeteilt. Die Berechnung erfolgt von der Wurzel bis zu den Blättern und ist mit einer Laufzeit von $O(n)$ effizient. Ein Beispiel für eine Treemap, die mit diesem Algorithmus erstellt wurde, ist in Abbildung 2.5 zu sehen.

Trotz der weiten Verbreitung des Ansatzes weisen Shneiderman und Johnson auf ein zentrales Problem nicht explizit hin: Der notwendige Abstand zwischen Rechtecken wird realisiert, indem von jedem Rechteck an allen vier Seiten ein Rand abgezogen wird. Dadurch ist die resultierende Fläche

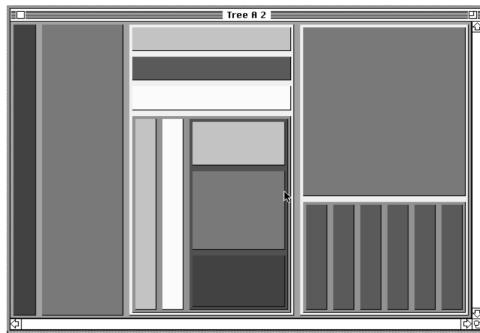


Abbildung 2.5: Beispiel für eine Treemap

oftmals nicht mehr proportional zu den zugehörigen Werten. Insbesondere sehr kleine oder langgestreckte Rechtecken können im Extremfall sogar komplett verschwinden. Dies verletzt die dritte Eigenschaft (strikte Proportionalität). In praktischen Anwendungsfällen, zum Beispiel wenn die Rechtecksfläche eine Kennzahl wie "Lines of Code" darstellen soll, können kleine Dateien oder Knoten durch die Ränder also komplett verloren gehen. Problematisch ist dies insbesondere, wenn eine weitere Dimension (wie beispielsweise die Testabdeckung) diese Elemente eigentlich hervorheben sollte, sie aber infolge der Skalierung durch Randabzug nicht mehr sichtbar sind.

Ein weiteres ästhetisches Manko liegt darin, dass die rekursive Unterteilung längliche oder stark verzerrte Rechtecksformen erzeugen kann. Dies widerspricht ebenfalls dem Anspruch übersichtlicher und ansprechender Visualisierung. Verschiedene Verbesserungsansätze wie etwa *Squarified Treemaps* begegnen diesem Problem und werden im Folgenden diskutiert (siehe Abschnitt 2.4.1).

In der Literatur besteht häufig Unklarheit über die Definition von 3D-Treemaps. Streng genommen bedeutet 3D-Treemap die rekursive Unterteilung eines Würfels (oder Quaders) in kleinere Volumina, wobei zum Wertvergleich das Volumen und nicht mehr die Fläche herangezogen wird. Gegenstand der vorliegenden Arbeit sind jedoch sogenannte *2.5D-Treemaps*, bei denen ein klassisches 2D-Treemap-Layout um eine zusätzliche dritte Dimension (zum Beispiel durch Extrusion nach oben) erweitert wird. Im Kontext dieser Arbeit ist mit einer "3D-Darstellung" stets dieses 2.5D-Modell gemeint.

2.4.1 Squarify-Algorithmus

Der Squarify-Algorithmus ist ein Verfahren zur Anordnung von Rechtecken in Treemaps, bei dem die Seitenverhältnisse der Rechtecke möglichst ausgeglichen gestaltet werden sollen. Ziel ist es, annähernd quadratische Rechtecke zu erzeugen, um die Nachteile herkömmlicher Treemap-Layouts zu überwinden, bei denen oft sehr schmale und langgezogene Rechtecke entstehen. Bruls et al. betonen dazu: "another problem of standard treemaps [is] the emergence of thin, elongated rectangles" [BHVWoo, S. 1]. Rechtecke mit

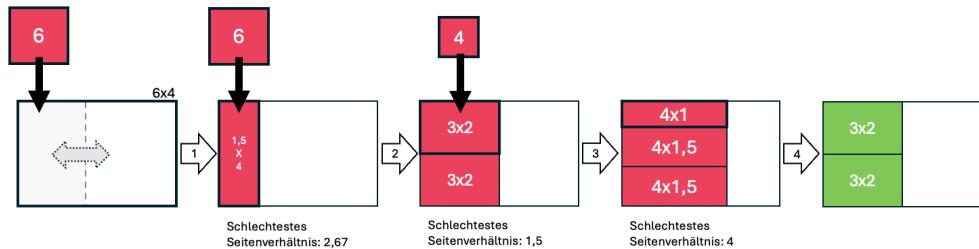


Abbildung 2.6: Beispiel für die erste Reihe des Squarify-Algorithmus. Die Reihe ist im linken großen Rechteck in hellgrau dargestellt, die kann nach rechts in der Breite variieren. Rot sind die Rechtecke, die in ihrer Breite und Höhe noch nicht festgelegt sind. Grün sind die Rechtecke, bei denen die Breite und Höhe neu festgelegt wurde.

möglichst ähnlichen Seitenlängen sind einerseits leichter wahrzunehmen, zu vergleichen sowie in ihrer Größe intuitiv abschätzbar.

Der Squarify-Algorithmus arbeitet rekursiv, indem er die zu verteilende Fläche schrittweise in Rechtecke aufteilt. Dabei wird stets angestrebt, das Verhältnis von Breite zu Länge der Rechtecke so nah wie möglich an einen Zielwert (idealerweise 1) zu bringen. Wie viele andere Treemap-Algorithmen erfolgt die Aufteilung vom Wurzelknoten her, indem die Fläche sukzessive in kleinere Teilflächen untergliedert wird, bis auf Blattknotenebene.

Im Folgenden wird der Algorithmus anhand eines Beispiels aus dem Originalartikel von Bruls et al. [BHWoo, S. 5] erläutert, wobei die Erklärung näher an einer praktischen Implementierung aus der d3-Bibliothek [D3ta] ausgerichtet ist.

Der Algorithmus füllt die zur Verfügung stehende Fläche stets in Reihen auf, wobei in jeder Reihe möglichst quadratische Rechtecke entstehen sollen. Das Einfügen erfolgt dabei iterativ: Das erste Rechteck wird eingefügt, dann das aktuelle Seitenverhältnis des Rechtecks berechnet. Dann wird das nächste Rechteck eingefügt. Wenn durch das Einfügen das Seitenverhältnis eines Rechtecks der Reihe schlechter wird, wird eine neue Reihe eröffnet.

In dieser Arbeit wird (abweichend von einigen Publikationen) die X-Koordinate als Breite und die Y-Koordinate als Länge bezeichnet, um Verwechslungen mit der Z-Komponente (Höhe im dreidimensionalen Raum) zu vermeiden.

Im Folgenden wird der Algorithmus zur Anordnung von Rechtecken anhand eines konkreten Beispiels erläutert. Es sollen Rechtecke mit den Größen 6, 6, 4, 3, 2, 2, 1 in ein übergeordnetes Rechteck mit den Abmessungen 6×4 eingesortiert werden.

Da das Rechteck, in welches eingefügt wird, breiter als hoch ist (Breite 6, Höhe 4), werden die Rechtecke einer vertikalen Reihe platziert. Diese vertikale Reihe hat immer die Höhe 4 und kann nach rechts in der Breite variieren (siehe das linke große Rechteck in Abbildung 2.6). Diese Reihe wird solange ergänzt, bis das schlechteste Seitenverhältnis (also das ungünstigste der erhaltenen Rechtecke) schlechter ist, als das schlechteste Seitenverhältnis im zuvorigen Schritt.

Zunächst wird in Schritt 1 das erste Rechteck mit einer Fläche von 6 Einheiten in die noch leere Reihe eingefügt. Das Seitenverhältnis dieses Rechtecks beträgt $1,5 \times 4$ (1,5 Einheiten breit und 4 Einheiten lang). Das schlechteste Seitenverhältnis aller Rechtecke in der aktuellen Reihe ist trivialierweise das des einzigen Rechtecks aktuell, also $4 : 1,5 \approx 2,67$ (wir teilen per Definition immer den größeren Wert durch den kleineren, dadurch ist 1 der optimale Wert, an den man sich von oben annähert, dadurch werden die vergleiche einfacher). In Schritt zwei wird das nächste Rechteck mit einer Fläche von 6 Einheiten in die Reihe *von oben* über das erste REchteck eingefügt. Dadurch wird die Reihe und auch das erste Rechteck breiter. In diesem Fall haben beide Rechtecke das gleiche Seitenverhältnis von 3×2 (3 Einheiten breit und 2 Einheiten lang). Das schlechteste Seitenverhältnis in der Reihe ist nun als $3 : 2 = 1,5$ zu berechnen. Das neue schlechteste Seitenverhältnis ist also besser als das vorherige, sodass die Reihe weiter gefüllt werden kann. Im nächsten Schritt soll ein Rechteck der Fläche 4 Einheiten eingefügt werden. Das hinzufügen dieses Rechtecks macht die Reihe erneut breiter. Es entsteht ein neues Schlechtestes Seitenverhältnis von $4 : 1 = 4$. Da das Verhältnis nun schlechter ist als das vorherige, wird die Reihe, so wie sie zuvor war, als abgeschlossen betrachtet. Das zuletzt eingefügte Rechteck (mit Fläche 4) wird daher nicht mehr in dieser Reihe, sondern in einer neuen Reihe eingesortiert (siehe Schritt 4).

Nachdem nun die erste Reihe abgeschlossen ist, wird eine neue Reihe eröffnet (siehe Abbildung 2.7). Das durch die zuvor erstellte Reihe verbliebene, zu füllende Rechteck hat die Abmessungen 3×4 (3 Einheiten breit und 4 Einheiten lang). Da die Reihe länger als breit ist. Durch das Einfügen des Rechtecks entsteht ein schlechtestes Seitenverhältnis von $3 : 1,33 \approx 2,25$. Das Einfügen des nächsten Rechtecks mit der Fläche 3 Einheiten verbessert das schlechteste Seitenverhältnis auf $2,33 : 1,29 \approx 1,8$ und ist damit besser als das vorherige. Das Einfügen des nächsten Rechtecks mit der Fläche 2 verschlechtert das schlechteste Seitenverhältnis auf $3 : 0,67 \approx 4,5$, sodass die Reihe abgeschlossen wird (Schritt 8).

Um den Ablauf besser zu verstehen ist unter diesem [Link \[Meh25b\]](#) ein Video zu erreichen, das für den algorithmus das Füllen der ersten zwei Reihen zeigt. Es ist zu erkennen, wie das einfügen der Rechtecke die Reihen breiter macht und wie das letzte Rechteck in der Reihe entfernt wird, wenn es das schlechteste Seitenverhältnis verschlechtert.

Dieser Prozess wird so lange fortgesetzt, bis alle Rechtecke in Reihen eingesortiert sind (siehe Abbildung 2.8).

Der Schritt, das jeweils schlechteste Seitenverhältnis in einer Reihe zu bestimmen, lässt sich rechnerisch effizient gestalten. Für ein Rechteck mit Fläche w_i in einer Reihe mit konstanter Länge l und Breite w sowie Gesamtfläche sV innerhalb der Reihe kann das maximale Seitenverhältnis durch folgende Umformungen vereinfacht berechnet werden:

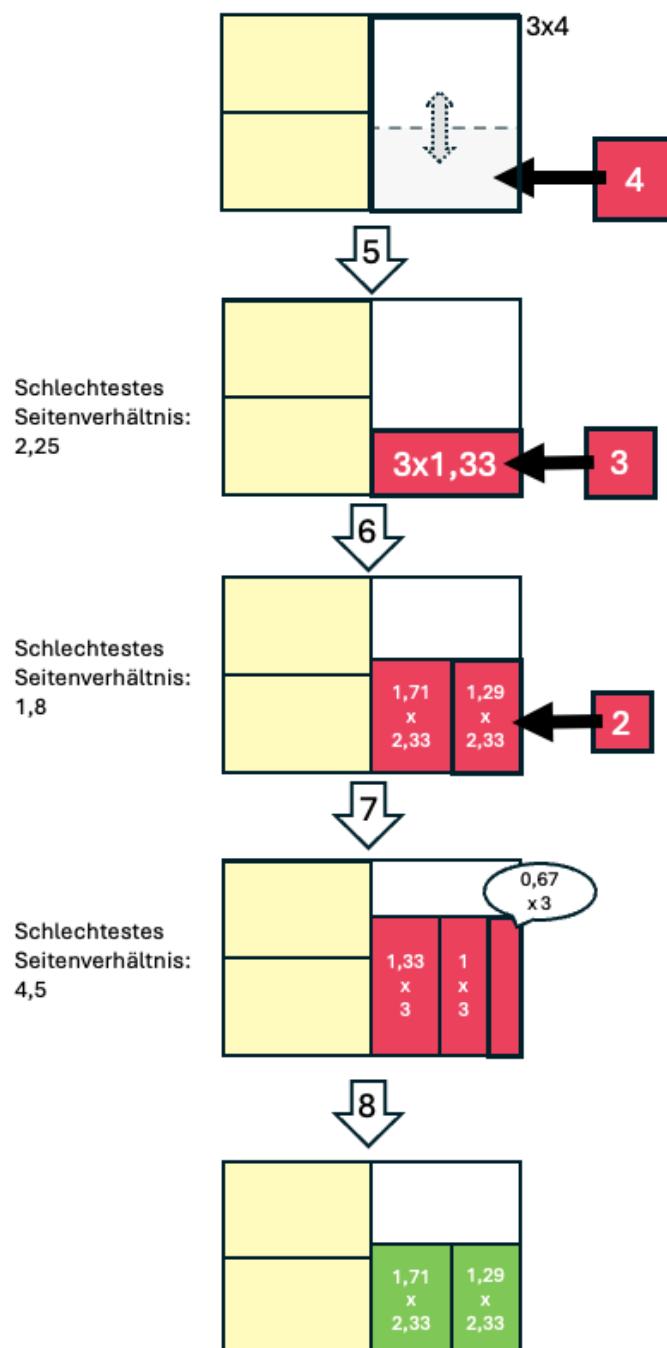


Abbildung 2.7: Beispiel für die zweite Reihe des Squarify-Algorithmus. Die Reihe ist im linken großen Rechteck in hellgrau dargestellt, die kann nach rechts in der Breite variieren. Rot sind die Rechtecke, die in ihrer Breite und Höhe noch nicht festgelegt sind. Gelb sind die Rechtecke die teil von zuvor abgeschlossenen Reihen sind. Grün sind die Rechtecke, bei denen die Breite und Höhe neu festgelegt wurde.

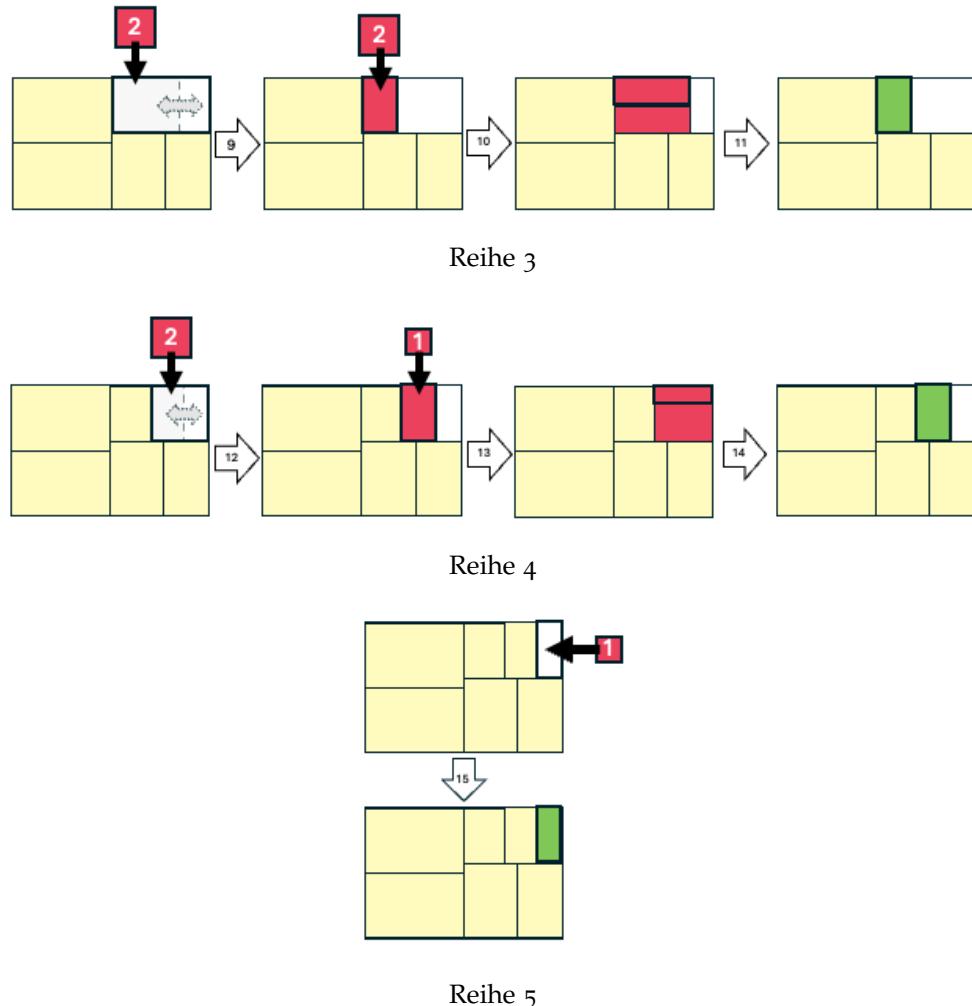


Abbildung 2.8: Beispiel für die letzten Reihen des Squarify-Algorithmus. Die Reihen sind im linken großen Rechteck in hellgrau dargestellt, die kann nach rechts in der Breite variieren. Rot sind die Rechtecke, die in ihrer Breite und Höhe noch nicht festgelegt sind. Gelb sind die Rechtecke die teil von zuvor abgeschlossenen Reihen sind. Grün sind die Rechtecke, bei denen die Breite und Höhe neu festgelegt wurde.

$$\frac{w_i}{l_i} = \frac{w_i \cdot l_i \cdot w^2}{l_i \cdot l_i \cdot w^2} \quad (2.1)$$

$$= \frac{w_i \cdot l_i \cdot w^2}{l_i^2 \cdot (\sum_{j=0}^n w_j)^2} \quad (2.2)$$

$$= \frac{w_i \cdot l_i \cdot w^2}{(l_i \cdot \sum_{j=0}^n w_j)^2} \quad (2.3)$$

$$= \frac{w_i \cdot l_i \cdot w^2}{(\sum_{j=0}^n l_i \cdot w_j)^2} \quad (2.4)$$

$$= \frac{w_i \cdot l_i \cdot w^2}{(\sum_{j=0}^n l_j \cdot w_j)^2} \quad \text{da } \forall i, j \in \{0, \dots, n\}, l_i = l_j \quad (2.5)$$

$$= \frac{V_i \cdot w^2}{sV^2} \quad (2.6)$$

Ebenso gilt analog $\frac{l_i}{w_i} = \frac{sV^2}{V_i \cdot w^2}$. Es genügt daher, für das jeweils größte und kleinste Rechteck in der Reihe den Wert zu berechnen und daraus das schlechteste Seitenverhältnis zu nehmen.

Während der Füllung einer Reihe bleibt w konstant und muss daher nur einmal berechnet werden; sV wird bei jedem neuen Rechteck in der Reihe aktualisiert.

Um den Algorithmus noch besser zu verstehen, hängen wir folgend den code an, so wie wir in dieser Arbeit als Grundlage für die Erweiterungen verwendet haben. Der Code ist in TypeScript geschrieben. Als Vorlage für diese Implementierung diente die JavaScript-Bibliothek [d3tb](#).

```

1  function squarifyNode(parent: SquarifyNode) {
2      // nodes sind die einzusortierenden Knoten
3      const nodes: SquarifyNode[] = parent.children
4      let row: SquarifyRow,
5          x0 = parent.x0,
6          y0 = parent.y0,
7          i = 0,
8          j = 0,
9          numberofChildren = nodes.length,
10         width: number,
11         length: number,
12         value = parent.value,
13         sumValue: number,
14         minValue: number,
15         maxValue: number,
16         newRatio: number,
17         minRatio: number,
18         alpha: number,
19         beta: number
20

```

```

21 // i ist der Index des aktuellen Knotens, der eingesortiert wird
22 while (i < numberOfWorkChildren) {
23     // Ein Schleifendurchlauf entspricht einer Reihe von Knoten
24     // Breite und Länge des noch zu belegenden Bereichs müssen nach
25     // jeder Reihe neu berechnet werden
26     width = parent.x1 - x0
27     length = parent.y1 - y0
28
29     // Suche den nächsten Knoten mit Wert und aktualisiere den
30     // sumValue
31     do {
32         sumValue = nodes[j++].value
33     } while (!sumValue && j < numberOfWorkChildren)
34
35     // minValue ist der kleinste Wert in der Reihe
36     // maxValue ist der größte Wert in der Reihe
37     // sumValue ist die Summe der Werte der Knoten in der aktuellen
38     // Reihe
39     minValue = maxValue = sumValue
40     // alpha ist der feste Faktor für die Berechnung des
41     // Seitenverhältnisses
42     alpha = Math.max(length / width, width / length) / (value *
43         aimedRatio)
44     // beta ist der variable Faktor für die Berechnung des
45     // Seitenverhältnisses
46     beta = sumValue * sumValue * alpha
47     // minRatio speichert das aktuelle schlechteste
48     // Seitenverhältnis in der Reihe
49     minRatio = Math.max(maxValue / beta, beta / minValue)
50
51     // Füge weitere Knoten hinzu, solange sich das Seitenverhältnis
52     // nicht verschlechtert
53     for (; j < numberOfWorkChildren; ++j) {
54         const nodeValue = nodes[j].value
55         sumValue += nodeValue
56         if (nodeValue < minValue) {
57             minValue = nodeValue
58         }
59         if (nodeValue > maxValue) {
60             maxValue = nodeValue
61         }
62         beta = sumValue * sumValue * alpha
63         // Berechne das neue schlechteste Seitenverhältnis nach dem
64         // Hinzufügen des Knotens
65         newRatio = Math.max(maxValue / beta, beta / minValue)
66
67         if (newRatio > minRatio) {
68             // Wenn das Seitenverhältnis schlechter wird, entferne
69             // den letzten Knoten und beende die Schleife
70             sumValue -= nodeValue
71             break
72         }

```

```

63     minRatio = newRatio
64 }
65
66 // Füllt die Reihe mit den ausgewählten Knoten und berechne
67 // deren Position und Größe
68 row = { name: parent.name, value: sumValue, dice: width <
69   length, children: nodes.slice(i, j) }
70 if (row.dice) {
71   treemapDice(row, x0, y0, parent.x1, value ? (y0 += (length
72     * sumValue) / value) : y0)
73 } else {
74   treemapSlice(row, x0, y0, value ? (x0 += (width * sumValue)
75     / value) : x0, parent.y1)
76 }
77
78 }

```

Listing 2.1: Vereinfachte und kommentierte Grundlage für den Algorithmus der in dieser Arbeit hauptsächlich verwendet wird

Obwohl sich das ursprüngliche Paper von Bruls et al. [BHVWoo] vor allem auf das Seitenverhältnis 1 als Idealwert konzentriert, erlauben viele Implementierungen, wie auch die implementierungen von d3.js [D3ta] auch die Annäherung an andere Zielwerte, beispielsweise an den Goldenen Schnitt [Lu+17]. In Abbildung 2.9 ist ein Beispiel für ein Layout zu sehen, dass durch den Squarify-Algorithmus generiert wurde, wobei die Rechtecke möglichst quadratisch werden sollen, so wie beim Original-Algorithmus von Bruls et al. [BHVWoo]. In Abbildung 2.10 sind die selben Werte zu sehen. Day layout wurde auch durch den Squarify-Algorithmus generiert, jedoch mit dem Ziel, dass die Rechtecke ein Seitenverhältnis von 5 anstreben. Es ist ein deutlicher Unterschied in den Darstellungen zu erkennen: Die Rechtecke sind deutlich langgestreckter, als im ersten Beispiel.



Abbildung 2.9: Beispiel für ein Squarify-Layout mit Annäherung an quadratische Rechtecke (durchschnittliches Seitenverhältnis 1,42)

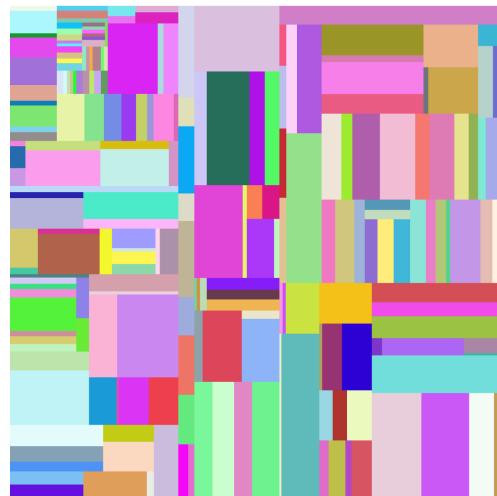


Abbildung 2.10: Beispiel für ein Squarify-Layout mit Annäherung an den Wert 5 (durchschnittliches Seitenverhältnis 2,79)

PROBLEMSTELLUNG

Die zentrale Herausforderung bei der Entwicklung von Stadtmetaphern zur Softwarevisualisierung besteht in der Festlegung eines geeigneten Layouts für die Visualisierung. Im Abschnitt 2.3 wurde das ursprüngliche Layout-Konzept der CodeCity-Analogie vorgestellt. Ein gemeinsames Grundprinzip dieser und verwandter Arbeiten ist, dass zunächst ein 2D-Layout erzeugt wird, das anschließend durch eine zusätzliche Metrik in die dritte Dimension extrudiert und schließlich durch weitere Metriken (z.B. Farbe, Transparenz) angereichert wird. In dieser Arbeit werden wir uns genau auf dieses Grundprinzip konzentrieren und untersuchen, wie sich verschiedene Layout-Algorithmen auf die Visualisierung von Code-Qualitätsmetriken auswirken. Die daraus entstehende Leitfrage lautet:

Welches Layout eignet sich am besten zur Visualisierung von Code-Qualitätsmetriken? Gibt es Ansätze, die Übersichtlichkeit und Verständlichkeit im Vergleich zu bestehenden Lösungen (insbesondere der CodeCity-Metapher) verbessern?

Daraus ergeben sich für diese Arbeit folgende konkrete Forschungsfragen:

1. Wie lassen sich bestehende Treemap-Layout-Algorithmen an das spezifische Anwendungsproblem dieser Arbeit anpassen? Welche Vor- und Nachteile resultieren daraus?
2. Lassen sich durch Analyse verwandter Arbeiten und bestehender Tools alternative Layouts identifizieren, die eine bessere Grundlage für die Visualisierung von Code-Qualitätsmetriken bieten?
 - a) Beispielhaft: Übertrifft das Sunburst-Layout das beste Treemap-Layout im Hinblick auf die Darstellung von Code-Qualitätsmetriken?
 - b) Beispielhaft: Bietet das Stack-Layout Vorteile gegenüber klassischen Treemap-Layouts für diesen Anwendungszweck?

3.1 Evaluationsrahmen

Um die angestrebten Visualisierungen eindeutiger zu evaluieren und in ihrem Ziel klarer zu definieren, orientieren wir uns an den fünf Dimensionen der Softwarevisualisierung nach Marcus Adrian et al. [MFMo3, S. 2]:

- **Tasks:** Warum wird die Visualisierung benötigt?
- **Audience:** Wer nutzt die Visualisierung?
- **Target:** Welche Datenbasis/Quelle wird abgebildet?
- **Representation:** Wie wird die Information dargestellt?
- **Medium:** Wo findet die Darstellung statt?

1. Warum ist die Visualisierung von Code-Qualitätsmetriken wichtig?

Die Visualisierung von Code-Qualitätsmetriken ermöglicht die Bewertung und das bessere Verständnis der Softwarequalität. Sie dient dazu, einen schnellen Überblick über komplexe Codebasen zu erhalten, Hotspots zu identifizieren und vertiefende Analysen effizient einzuleiten. Besonders wichtig ist es dabei, Zusammenhänge zwischen verschiedenen Metriken auf einen Blick erfassbar zu machen, was in rein tabellarischer oder isolierter Darstellung kaum möglich ist. Das Hauptziel besteht darin, das abstrakte Konzept der Codequalität *greifbar* und anschaulich zu machen.

2. Wer ist die Zielgruppe der Visualisierung?

Die Visualisierung richtet sich primär an Personen ohne tiefgehende Kenntnisse der Codebasis und ohne Programmierkenntnisse. Trotzdem kann die Visualisierung aber auch Entwicklerinnen und Entwicklern helfen, die sich neu in ein Projekt einarbeiten oder Teams, die systematisch die Softwarequalität verbessern wollen, in der Kommunikation unterstützen.

3. Was ist die Datenquelle?

Grundlage sind hierarchische Code-Qualitätsmetriken, die automatisiert aus dem Source Code extrahiert werden. Wobei die Metriken auf File-Basis erstellt werden. Dieser Vorschlag stammt ursprünglich von Code City. Er bietet eine gute Granularität, um die Struktur und die Qualität von Software zu beurteilen [WLo7]. Jeder Knoten der Hierarchie (Node) besitzt dabei einen Namen und entweder eine Liste von Kindknoten (children) oder einen Wert (value), der die Metrik repräsentiert (siehe Listing 3.1). Diese Struktur ermöglicht es, komplexe Softwareprojekte in übersichtliche Hierarchien zu zerlegen, die dann visualisiert werden können.

```
"node": {
    "name": string,
    "children": List[Node] | "value": number
}
```

Listing 3.1: Schema einer Node

Genauer werden wir in Abschnitt 6 auf die Eingabedaten eingehen.

4. In welchem Medium soll die Visualisierung erfolgen?

Die Visualisierung ist für die digitale Darstellung auf herkömmlichen Bildschirmmedien konzipiert. Im vorliegenden Prototyp wird eine Umsetzung im Webbrowser mittels TypeScript und Three.js realisiert. Die Übertragung der Methoden auf andere Programmierumgebungen ist möglich.

5. Wie erfolgt die Darstellung?

Das Visualisierungskonzept lehnt sich an den in den Grundlagen erläuterten Stadtmetapher-Ansatz an (Abschnitt 2.3). Im Mittelpunkt dieser Arbeit steht das zweidimensionale Layout der Knoten. Die Gestaltung dieses Layouts wird jedoch eingeschränkt, dadurch, dass Fläche, Höhe und Farbgebung (auch Schattierung, Struktur oder Transparenz) zur Darstellung verschiedener Metriken dienen. Maßnahmen wie das Einfärben von verschiedenen Ordnern oder Modulen, zur besseren visuellen Unterscheidung fallen für die Visualisierung in dieser Arbeit weg. Stattdessen wird der Fokus auf die geschickte Platzierung, Wahl von Abständen und Beschriftung gelegt.

Aus dem hier definierten Rahmen und den in den Grundlagen beschriebenen Grundlegenden Anforderungen (Abschnitt 2.2.1) leiten sich für die, im Rahmen dieser Arbeit entwickelte, Visualisierung folgende Anforderungen ab:

- **Informationsgehalt und effiziente Nutzung des Platzes:** Möglichst viel relevante Information soll bei minimalem Flächenverbrauch dargestellt werden.
- **Niedrige visuelle Komplexität und hohe Verständlichkeit:** Das Layout muss klar und intuitiv lesbar bleiben, um Überforderung zu vermeiden.
- **Skalierbarkeit:** Auch umfangreiche Softwaresysteme müssen übersichtlich visualisierbar sein.
- **Korrelation mit dem Code:** Eine gute Zuordnung zwischen Visualisierung und Source Code muss gewährleistet sein, um Rückschlüsse auf die Softwarestruktur zu ermöglichen.
- **Stabilität gegenüber Änderungen:** Nachrangig, aber erwünscht ist, dass die Visualisierung auf Änderungen im Code robust reagiert, um zeitliche Vergleiche zu ermöglichen, ohne dass der Betrachter sich neu orientieren muss.

3.2 Das Treemap-Problem

Durch die nähere Erläuterung des Treemap-Algorithmus (siehe Abschnitt 2.4) wurde gezeigt, dass klassische Treemap-Algorithmen grundlegende Schwächen aufweisen, insbesondere wenn Abstände (*Margins*) zwischen den Knoten dargestellt werden sollen. Das Grundproblem, das von Johnson und Shneiderman [JS91] adressiert wurde, ist bereits ohne Abstände NP-Hard [BHWoo, S. 3], und zusätzliche Abstände verkomplizieren die Problematik noch weiter.

Wesentliche Schwierigkeiten bei der Integration von Abständen in das Layout ergeben sich durch:

- Die abgezogenen Flächen für Abstände führen dazu, dass die verbleibende Knotenfläche nicht mehr proportional zum Wert des Knotens ist.
- Bei kleinen Knoten können durch Abzüge von Minimalabständen einzelne Knoten ganz verschwinden, wenn ihre minimale Länge oder Breite kleiner als der geforderte Abstand wird.

In den Abbildungen 3.1, 3.2 und 3.3 sind verschiedene Treemap-Layouts zu sehen, die mithilfe des Squarify-Algorithmus (siehe Abschnitt 2.4.1) erzeugt wurden. Die zugrundeliegenden Strukturen und Metriken wurden dabei händisch erstellt, um typische Probleme bei der Flächenproportionalität und Sichtbarkeit in Treemaps zu verdeutlichen (siehe Anhang A.1).

In Abbildung 3.1 sind alle Knoten sichtbar, und die Flächen allen Knoten sind proportional zu dem jeweiligen Wert. Dadurch ergibt sich eine exakte Flächenproportionalität zwischen Wert und Darstellung.

In Abbildung 3.2 ist zwar weiterhin jeder Knoten sichtbar, jedoch sind die Flächen nicht mehr exakt proportional zu den Werten der Knoten. Beispielsweise besitzt der große Knoten 3 mit dem Wert 3000 nur noch eine Fläche von etwa 2600, was ein Flächen-Wert-Verhältnis von ungefähr 0,9 ergibt. Der kleine Knoten oben links (Knoten 5) mit dem Wert 30 weist hingegen nur noch eine Fläche von etwa 5 auf, entsprechend einem Verhältnis von ca. 0,2. Knoten 2 hat eine Fläche von etwa 17, er erscheint also, obwohl er den selben Wert hat wie Knoten 5, deutlich größer. Dies verdeutlicht, dass die Abstände die Flächenproportionalität erheblich beeinträchtigen.

In Abbildung 3.3, bei einem Abstand von 10, werden einige Knoten, zum Beispiel wie etwa der zuvor oben links gelegene Knoten 5 mit Wert 30, gar nicht mehr dargestellt, da ihre berechnete Breite kleiner als der vorgegebene Abstand ist. Damit sind relevante Informationen in der Visualisierung nicht länger sichtbar.

Diese Beispiele verdeutlichen die Auswirkungen gewählter Abstandsparameter auf Proportionalität und Lesbarkeit von Treemaps.

Das zentrale Dilemma entsteht dadurch, dass die Kernannahme der Treemap-Algorithmen verletzt wird: Die benötigte Fläche aller Knoten einer Hierarchieebene muss vor deren Anordnung bekannt sein. Wird jedoch Fläche für Abstände benötigt, ist vorab unklar, wieviel Raum tatsächlich für jeden Knoten zur Verfügung steht. Die effektive *Belegungsfläche* hängt erheblich von der jeweiligen Anordnung der Knoten ab.

Diese Problematik illustriert Abbildung 3.4. Zwei Rechtecke identischer Fläche (16 Einheiten) benötigen im Layout mit je einem Abstand von 1 eine unterschiedlich große Gesamtfläche, je nachdem, ob sie quadratisch (z.B. 4×4) oder länglich (z.B. 2×8) angeordnet sind; die zusätzliche Fläche für Abstände variiert also erheblich.

Dadurch entsteht ein Zirkelschluss: Das endgültige Layout kann erst berechnet werden, wenn die tatsächlich benötigte Knotenfläche (inklusive Abstände) bekannt ist. Diese Fläche hängt aber wiederum vom Layout ab. Eine triviale Lösung existiert hierfür nicht, weshalb in dieser Arbeit verschiedene Ansätze untersucht werden, um dieses Problem zu lösen (siehe Abschnitt 7).

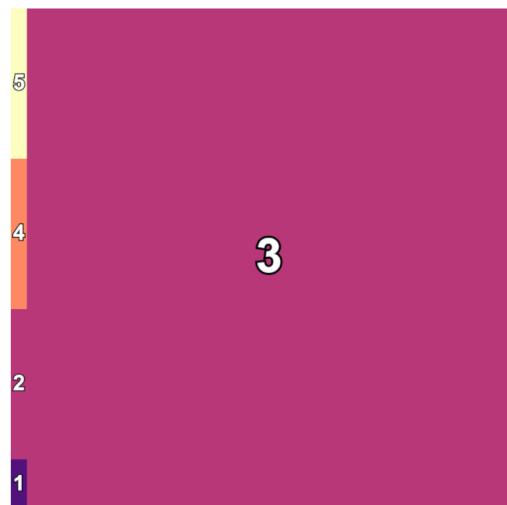


Abbildung 3.1: Treemap-Layout, generiert mit dem Squarify-Algorithmus (Abstand 0).

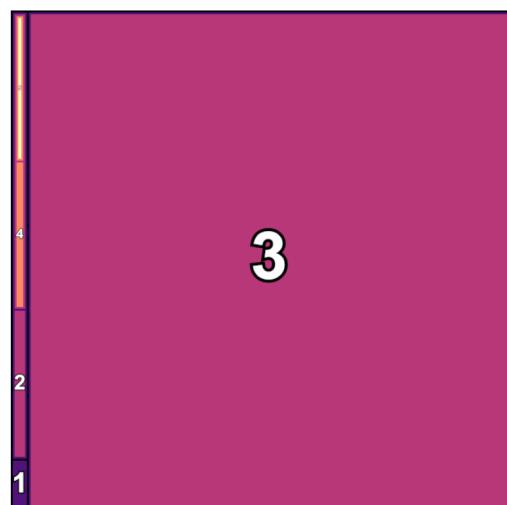


Abbildung 3.2: Treemap-Layout, generiert mit dem Squarify-Algorithmus (Abstand 5).

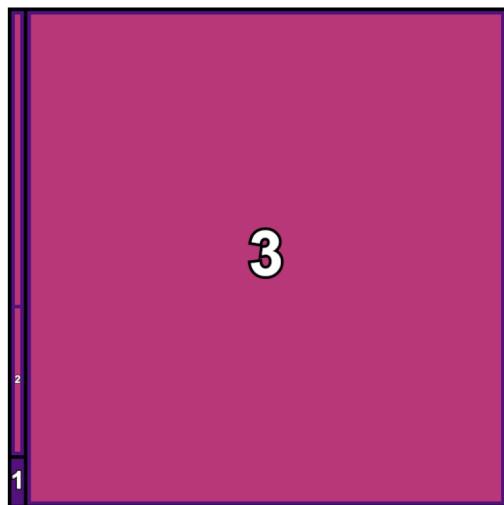


Abbildung 3.3: Treemap-Layout, generiert mit dem Squarify-Algorithmus (Abstand 10).

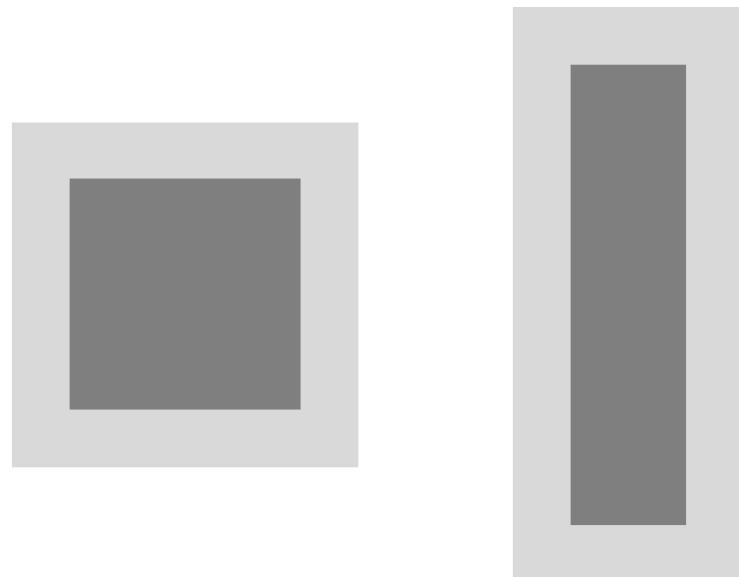


Abbildung 3.4: Zwei Rechtecke der Fläche 16 (dunkelgrau) mit umgebendem Abstand 1 (hellgrau): Links (4×4) mit Gesamtfläche 25 (5×5), rechts (2×8) mit 40 (4×10). Der jeweils notwendige Platz für Abstände hängt vom Seitenverhältnis ab.

VERWANDTE ARBEITEN

Im Folgenden werden Forschungsarbeiten vorgestellt, die sich mit Fragestellung beschäftigen, die den beiden zuvor vorgestellten Problemstellungen (siehe Abschnitt 3) ähnlich sind. Zunächst wird ein Überblick über Arbeiten gegeben, die sich mit dem Vergleich oder Vorstellung mehrerer Visualisierungstechniken beschäftigen. Anschließend werden Arbeiten vorgestellt, die sich speziell mit dem Treemap-Problem beschäftigen, also dem Problem von verschwindenden Knoten.

4.0.1 Vergleich von Visualisierungstechniken

Die Arbeit von Pacione et al. (*A comparative evaluation of dynamic visualisation tools* [PRWo3]) vergleicht fünf dynamische Visualisierungstools bezüglich ihrer Eignung für das Softwareverständnis und das Reverse Engineering. Die Autoren verfolgen das Ziel, Gründe dafür zu identifizieren, weshalb der Einsatz dieser Werkzeuge außerhalb der Forschung bislang nur begrenzt erfolgt, obwohl sie großes Potenzial für die Analyse komplexer Softwaresysteme bieten. Die Studie macht deutlich, dass keines der untersuchten Tools alle Anforderungen an Softwareverständnis vollständig abdeckt. Insbesondere wird auf Defizite in der Abbildung von Design-Patterns und Hotspot-Analysen hingewiesen, was unter Anderem im Kontext der Evaluation in dieser Arbeit untersucht werden soll.

Einen Überblick im Bereich der dreidimensionalen Softwarevisualisierung gibt die Arbeit von Teyseyre et al. (*An overview of 3D software visualization* [TCo8]). Die Autoren präsentieren verschiedene Visualisierungsansätze und Werkzeuge. Dabei werden folgende Kategorien differenziert:

- **Graphbasierte Ansätze:** 3D-Knoten-Kanten-Diagramme mit Fokus auf die Darstellung von Klassenbeziehungen
- **Baumstrukturen:** Dreidimensionale Variationen wie *Cone Trees*, *Information Cubes* (im Grund 3D Treemaps) oder *hierarchical nets* (siehe Abbildung 4.1) zur hierarchischen Darstellung von Softwareelementen
- **Abstrakte Geometrische Darstellungen:** Nutzung von verschiedenen Formen zur Darstellung von verschiedenen Metriken und Strukturen. (Beispiel in Abbildung 4.2)

Sie zeigen auch auf, dass einige Ansätze Metaphern wie Städte oder Landschaften aus der echten Welt nutzen. Interessant ist, dass von den 22 analysierten Tools nur vier für das Management oder Stakeholder ohne spezielle Entwicklungskompetenzen geeignet sind. Von diesen vier visualisiert zudem lediglich eines dieser Werkzeuge tatsächlich den Quellcode. Die anderen drei Tools konzentrieren sich auf die Visualisierung von Anforderungen.

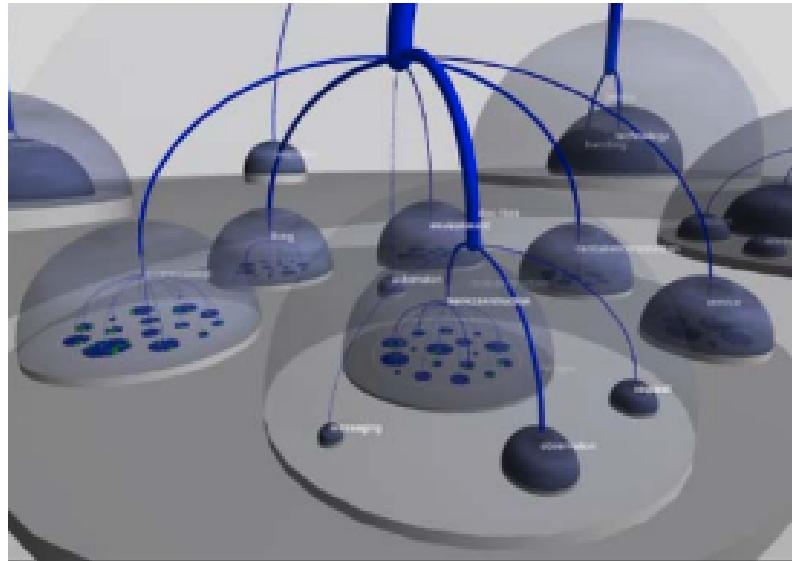


Abbildung 4.1: Beispiel für ein hierarchisches Netz in 3D. Die hierarchische Struktur wird durch Kreise, Halbkugeln und deren Verbindungen dargestellt. [TC08, S. 6]

Zudem wird ein Mangel an empirischen Nutzungsstudien im 3D-Bereich konstatiert, was den Status und die Praxistauglichkeit der Ansätze weiter einschränkt. [TC08] In dieser Arbeit wird daher untersucht, ob die in dieser Arbeit vorgestellten Ansätze auch für Nutzer ohne tiefere technische Kenntnisse geeignet sind.

Auch die Arbeit von Caserta und Zendra (*Visualization of the static aspects of software: A survey* [CZ10]) gibt einen Überblick über etablierte und aktuelle Visualisierungstechniken und ordnet diese unterschiedlichen Analyseebenen zu:

- **Quellcode-Ebene:** Zeilenorientierte Visualisierungen von einzelnen Dateien oder Klassen
- **Klassen-/Mittel-Ebene:** Darstellungen der Strukturen oder Relationen von Dateien oder Klassen
- **Architekturebene:** Globale Übersichten über Organisation und Qualitätsattribute, etwa durch Darstellung von Vererbungen und Aufrufstrukturen

Für die Architekturebene, die im Fokus dieser Arbeit steht, werden insbesondere folgende Visualisierungstypen diskutiert: Tree/Node-Link-Diagramme (wie Bereits in der Grundlagen gezeigt: bei großen Systemen schnell unübersichtlich), Treemaps und deren Varianten (Circular, Sunburst, Voronoi), die hierarchische Strukturen platzsparend repräsentieren und in 2D sowie 3D verfügbar sind. Sie stellen unter anderem folgende Probleme dieser Visualisierungen fest: Schlechte Nutzbarkeit, kognitive Überlastung, Desorientierung in dreidimensionalen Darstellungen sowie eine geringe Verbreitung

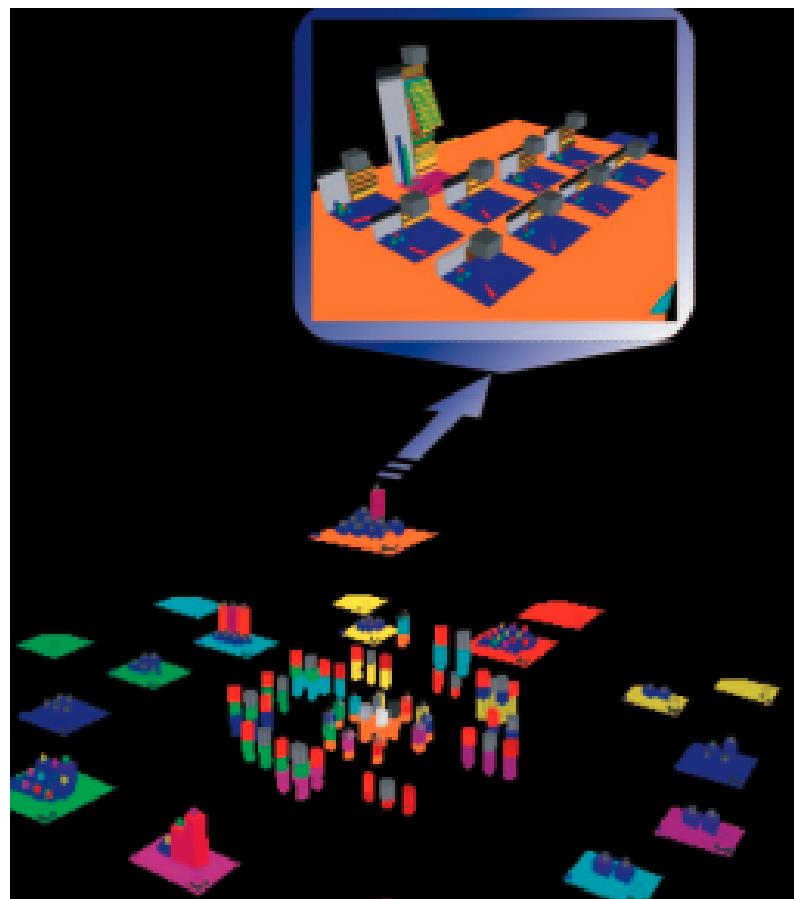


Abbildung 4.2: Beispiel für eine abstrakte geometrische Darstellung von Software.
[TC08, S. 7]

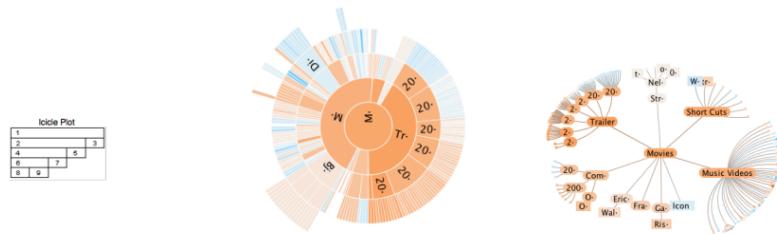


Abbildung 4.3: Beispiele für *Icicle Plots* (links), *Sunburst-Diagramme* (mitte) und *hyperbolische Treelayouts* (rechts). [Kha+12, S. 4]

praxistauglicher Tools, da viele existierende Lösungen im Prototypstatus sind.

Die Übersicht von Khan et al. (*Visualization and evolution of software architectures* [Kha+12]) stellt ebenfalls verschiedene Visualisierungstechniken vor. Sie unterscheiden zwischen hierarchischen und beziehungsorientierten Visualisationstechniken. Bei den hierarchischen Ansetzen werden platzfüllende Treemaps vorgestellt, die aufgrund ihrer Eigenschaften vor allem für große Hierarchien geeignet sind, wobei Sie hervorheben, dass Einschränkungen hinsichtlich der Visualisierung multipler Metriken und der Anschaulichkeit interner Strukturen bestehen, was in dieser Arbeit angegangen wird. Ergänzend nennen die Autoren auch Icicle Plots, Sunburst-Diagramme und hyperbolische Bäume (siehe Abbildung 4.3).

In *Software visualization tools: survey and analysis* [BK01] von Bassil und Keller führen eine Umfrage durch, in der sie untersuchen, was Nutzern bei Softwarevisualisierungstools wichtig ist. Es zeigte sich, dass insbesondere hierarchische Repräsentationen, Benutzerfreundlichkeit und Skalierbarkeit bei großen Softwaresystemen von den Nutzern als entscheidend bewertet werden. Auch hier ist auffällig, dass speziell Experten befragt werden und sich auch die meisten untersuchten Werkzeuge primär an Experten und weniger an Endanwender wie Softwarekunden oder Management richten.

Eine detaillierte Übersicht verschiedener Treemap-Layout-Algorithmen geben Scheibel et al. in ihrer Arbeit *Survey of treemap layout algorithms* [SLD20]. Die Autoren unterscheiden Treemap-Algorithmen anhand der Art der Aufteilung (packing vs. splitting). Splitting ist die klassische Treemap-Variante, bei der die Fläche geteilt wird. Packing-Ansätze hingegen platzieren die Rechtecke so, dass sie möglichst wenig Platz verschwenden. Beide Ansätze erzeugen leicht Verschiedene Layouts (siehe Abbildung 4.4).

Zudem unterscheiden sie verschiedene layouts anhand der Layout-Form (z.,B. rechteckig, kreisförmig, konvex, nicht-konvex) sowie der Referenzraum-Dimension (2D vs. 3D). Für die in dieser Arbeit betrachteten Anwendungsfälle stehen rechteckige 2D-Splitting-Layouts im Vordergrund (siehe Abschnitt 3 und 2.4) Wir stellen fest, dass von 81 analysierten Ansätzen sind 54 rechteckige und 58 splitting-basiert sind, was zeigt, dass dies das präferierte Treemap verfahren ist. Neben klassischen Verfahren (Slice-and-Dice, Squarified, Strip) werden auch Polygon-, Voronoi- und Circular-Treemaps sowie spezialisierte Layouts beschrieben, etwa für Aspektverhältnis-Optimierung oder

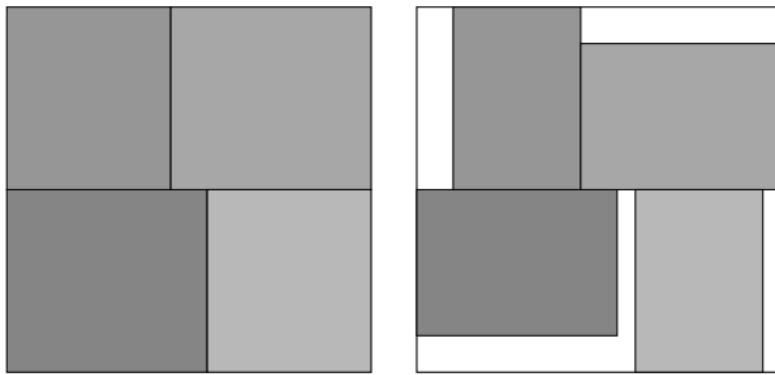


Abbildung 4.4: Beispiel für Layouts die durch Splitting (links) und Packing (rechts) Algorithmen erzeugt wurden. Es ist zu erkennen, dass bei Packing-Algorithmen mehr Platz zwischen den Knoten entsteht. [SLD20, S. 3]

Ordnungserhaltung. Die große Vielfalt verdeutlicht die Notwendigkeit einer gezielten anwendungsspezifischen Auswahl des Algorithmus.

Zusammenfassend zeigen die analysierten Arbeiten eine ausgeprägte Diversität an Visualisierungsansätzen, jedoch auch eine Diskrepanz zwischen Forschung und Praxis. Besonders mangelt es an expliziter Unterstützung für Endnutzer und Stakeholder, an der Integration empirischer Studien zur Nutzbarkeit sowie an leistungsfähigen, intuitiven Tools für den produktiven Einsatz. Die in dieser Arbeit fokussierten Layout- und Visualisierungstechniken orientieren sich daher an den identifizierten Herausforderungen und dem Bedarf nach effektiven Architekturendarstellungen insbesondere für nicht-technische Anwender.

Alles in allem: 1. vor allem auf experten ausgerichtet 2. wenig empirische Studien und Evaluation und vergleich 3. wenig umsetzung der tools in der praxis 4. viele arbeiten, die nur vorstellen und nicht wirklich vergleichen

4.1 Treemap Problem verwandte Arbeiten

Wir suchen hierfür speziell nach layouts die abstände haben. die meisten treemap layouts haben keine abstände, wodurch bei der extrusion ins drei dimensionale und ohne farbgebung die Struktur der Hierarchie nicht mehr erkennbar ist. Deswegen wird speziell nach treemap layouts gesucht, die abstände haben und die Struktur der Hierarchie darstellen.

Aktuell gibt es nur eine arbeit die dieses Problem explizit anspricht und zwar [LFo8].

[LFo8]: Hao Lü and James Fogarty stellten in ihrem Paper *Cascaded Treemaps: Examining the Visibility and Stability of Structure in Treemaps*[LFo8] fest: “an important limitation of treemaps is the difficulty of discerning the structure of a hierarchy”[LFo8, S. 1] Das stellt im Grunde das Problem von Treemaps dar, welches auch algorithmisch nicht einfach gelöst werden kann (siehe Abschnitt 3.2). Die Idee ist anders als bei Nested Ansätzen die Kindknoten nicht einfach in den Elternknoten zu zeichnen, sondern sie leicht versetzt

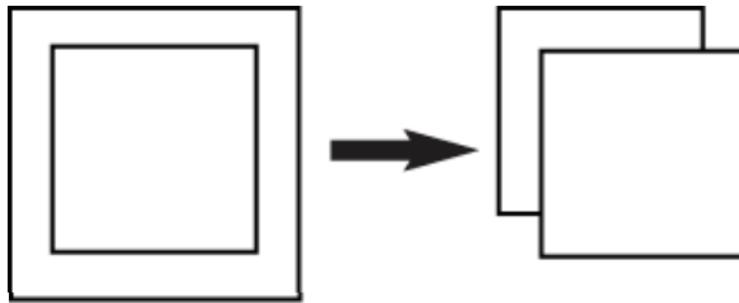


Abbildung 4.5: Links beispielhaft der Nested treemaps Ansatz, bei dem der Kindknoten einfach in dem Elternknoten gezeichnet wird. Rechts der Cascaded Treemap Ansatz, bei dem der Kindknoten als rechteck leicht versetzt nach rechts unten *über* dem Elternknoten gezeichnet wird. Abbildung aus [LFo8, S. 3].

über dem Elternknoten zu zeichnen (siehe Abbildung 4.5). Dadurch soll weniger Platz verloren gehen und es entsteht ein leichter 3D-Effekt.

Sie stellen in ihrem Paper auch fest, dass manche Knoten verschwinden können, da der Platz der für Beschriftung und Abstände benötigt wird, beim Layoutschritt nicht berücksichtigt werden kann. Sie stellen einen Zwei Schritten Ansatz vor, der im ersten Schritt mit den squarify Algorithmus [BHVWoo] das Layout erstellt. Im zweiten Schritt wird dann die Größe, aber nicht die Platzierung der Knoten angepasst, indem der Abstand und Platz für Beschriftung berücksichtigt wird. Das Problem von verschwindenden Knoten wird dadurch nicht komplett gelöst, aber Knoten verschwinden nur noch, wenn der Platz für die Beschriftung und den Abstand größer ist als der zur Verfügung stehende Platz. Die Autoren geben leider keinen Pseudo-Code für die exakte Implementierung an, weshalb es schwer ist die genaue Berechnung nachzuvollziehen und Unterschiede und Vor- und Nachteile zu den Implementierungen in dieser Arbeit aufzuzeigen. Sie beschreiben ihren Schritt wie folgt:

The function then computes how much vertical space is needed for offsets [...]. The remaining space is for the content of the treemap, and so the layout function gives each side of the split the space computed as necessary for [...] offsets as well as a portion of the remaining content space based on the relative weights of nodes on each side of the split. The layout procedure is then ready to recurse on both sides of the split, as it knows how much space will be used by [...] offsets and has ensured that the remaining space is appropriately divided by node weight. [LFo8, S. 6]

(FÜR MICH: das ist quasi simple-increase mit scaling) Sie berechnen also die Fläche, für jeden Knoten neu. An jeder Teilungs-Kante, die Kante an der ein Knoten geteilt wurde (entweder horizontal oder vertikal) - im Grunde werden sich immer die Reihen angeschaut. Wird dann der Platz berechnet, der für die Abstände benötigt wird. Anschließend wird der Platz für die Knoten

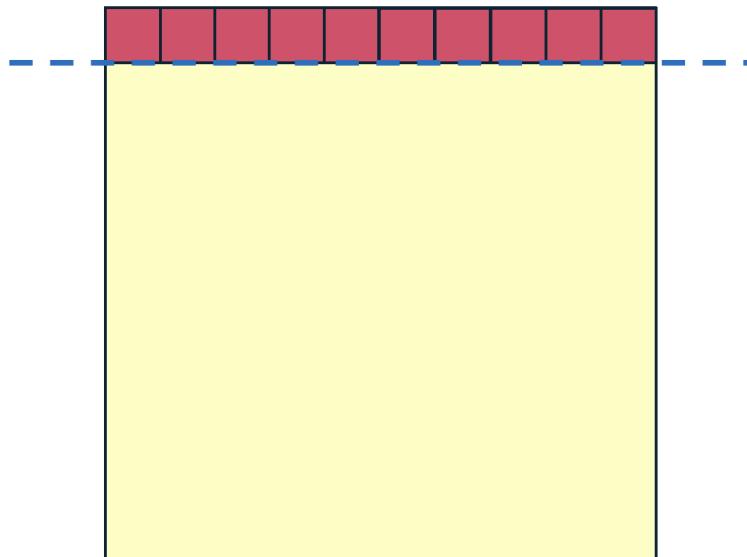


Abbildung 4.6: Beispiel für eine für den Cascaded Treemap Algorithmus schlechte Rechteck-Konstellation

berechnet, indem der Platz für die Abstände von der Gesamtfläche abgezogen wird. Dadurch wird sichergestellt, dass jeder Knoten genug Platz für die Abstände hat. Dennoch kann es passieren, dass Knoten verschwinden, wenn der Platz für die Abstände größer ist als der Platz, der für die Knoten zur Verfügung steht. Obwohl der Code nicht verfügbar ist, wird doch allein bei der Beschreibung ein Nachteil deutlich: Es wird an jeder Reihen-Kante nur der Platz mit in die Berechnung einbezogen, der senkrecht zu der Kante steht. In Abbildung 4.6 ist deutlich, dass der Algorithmus für den Bereich über von der Kante banauso viel Platz für die Abstände berechnet, wie für den Bereich unter der Kante, weil eben nur der Platz senkrecht zur Kante betrachtet wird. Und das obwohl offentlicht die Roten Rechtecke zusammen viel Mehr platz für die Abstände benötigen würden, als das Gelbe Rechteck alleine. Das Grundlegende Problem, welches wir zuvor beschrieben haben (siehe Abschnitt 3.2), ist also nicht gelöst.

Interessant ist, dass die Autoren keine Verbesserung in der Gewicht zu Größe relation feststellen konnten. (Ich vermute, dass das daran liegt, dass die Berechnung der neuen Größen für die Knoten nicht optimal ist) Eine Verbesserung konnten sie nur bei relativ kleinen Knoten feststellen, was auch klar ist, da bei den ansätzen zuvor die Abstände (da diese ja absolut sind und bei jedem knoten gleich) die Größe von kleinen Knoten viel stärker beeinflussen, als die Größe von großen. Sie vermuten auch, dass das an Ihrem Ansatz speziell liegen könnte und fordern noch weitere research in diesem Bereich. Sie vermuten außerdem, dass speziell bei Tiefen Hierarchien, die Abweichung von Größe zu Gewicht größer wird.

Eine ähnliche implementierungen untersuchen wir in Abschnitt HIER EINFÜGEN und zeigen genau auf, warum dieser Ansatz große Schwächen hat. bzw Am ehesten ist dieser Ansatz wahrscheinlich mit dem Simple-Increase



Abbildung 4.7: Beispiel für eine Cushion Treemap [VWW99, S. 4]

Ansatz mit Scaling und festen Knoten zu vergleichen, nur mit dem Unterschied, dass die kompletten abstände betrachtet werden und diese betrachtung nur einmalig nach dem ersten squarify Layout Schritt durchgeführt wird (was auch in dem casced paper angemerkt wurde, dass das einmalige berechnen der Abstände eine mögliche Optimierung ihres Ansatzes wäre [LFo8, S. 6]).

Die Autoren des ursprungs squarify Algorithmus [BHVWoo] stellen in einem anderen Paper [VWW99] eine Idee vor, um Struktur ohne Änderung des Layouts darzustellen und zwar mit Schatten (siehe Abbildung 4.7). Dabei bekommt jeder Knoten, egal ob Eltern- oder Kindknoten, einen Innenröhenschatten, wodurch die Struktur der Knoten sichtbar wird.

Peter Demian und Renate Fruchter stellen die Idee vor dass dickere outlines he höher der Knoten ist auch die Struktur verdeutlichen können.

Nicholas Kong et al schlagen vor verschiedene Umriss dicken zu nutzen um die Struktur der Knoten darzustellen.[KHA10] (siehe Abbildung ??).

5

LAYOUT SUCHE

Das Ziel dieses Abschnitts ist es 2D-Layouts zu finden, die geeignet sind, um extrudiert zu werden und Software-Qualitätsmetriken in Form der in Abschnitt 3 definierten Form in 2.5D zu visualisieren. Alternativ können auch direkt 2.5D Visualisierungen gefunden werden, die geeignet sind, um Software-Qualitätsmetriken wie in der definiten Form zu visualisieren. In diesem Abschnitt stellen wir zunächst die Methodik vor, die wir bei der Recherche verwendet haben. Anschließend stellen wir die Ergebnisse der Recherche vor.

5.1 *Methodik*

Die hier verwendete Methodik orientiert sich an der Methodik der systematischen Literaturrecherche, wie sie in *Procedures for performing systematic reviews*[Kit+04] beschrieben wird, ist aber leicht angepasst, da es vor allem um das finden und einordnen von Layouts geht und nicht direkt um das Bewerten und vergleichen von verschiedenen Arbeiten.

1. *Forschungsfrage*

Die Problemstellung dieser Arbeit an der hier gearbeitet wird ist bereits definiert:

Lassen sich durch Analyse verwandter Arbeiten und bestehender Tools (zum Treemap Layout) alternative Layouts identifizieren, die eine bessere Grundlage für die Visualisierung von Code-Qualitätsmetriken bieten? (siehe Abschnitt 3)

Die Konkrete Forschungsfrage bzw. das Ziel dieser Recherche leitet sich aus dieser Problemstellung ab: Unsere Recherche hat das Ziel, Layouts zu finden, die geeignet sind, um Software-Qualitätsmetriken in 2.5D zu visualisieren. Wir wollen dabei nicht nur spezifisch nach 2.5D visualisierungen suchen, sondern auch nach 2D-Layouts, die geeignet sind, um in 2.5D extrudiert zu werden.

2. *Suchstrategie*

Wir wollen sowohl wissenschaftliche Arbeiten als auch bestehende Tools und Ansätze finden, die sich mit der Visualisierung von Software-Qualitätsmetriken beschäftigen und analysieren, ob diese Ansätze geeignet sind, um auf unser spezifische Problemstellung übertragen zu werden.

Für die Suche nutzen wir zunächst Google Scholar wir wollen folgende Suche nutzen:

Wir wollen paper suchen, die sich mit der Visualisierung von Software-Qualitätsmetriken beschäftigen. Wir teilen unsere Suche in zwei Teile auf: Der fordere Teil bezieht sich auf die Visualierung. Und Der Hintere teil soll sicher gehen, dass es sich um Software-Qualitätsmetriken handelt, dafür muss auf jeden fall das wort software vorkommen und dann entweder "qualityöder "metrics". Beides muss im Paper vorkommen, um in unserer suche aufzutauchen. Daraus ergibt sich folgende Suchanfrage: ("3DÖR "2.5DÖR "visualizationÖR "treemapÖR "hierarchical data") AND (software AND ("qualityÖR "metrics"))

Zur suche nutzen wir die erweiterte Suche von Google Scholar, um möglichst viele relevante Ergebnisse von verschiedenen Journalen und Anbietern zu finden. Das Problem bei der erweiterten suche von Google scholar ist, dass sie sehr limitiert ist und keine verschachtelten Suchanfragen unterstützt [Sch23]. Deshalb teilen wir die Suche in zwei Teile auf: "3DÖR "2.5DÖR "visualizationÖR "qualityÖR "metrics AND software AND "treemapÖR "hierarchical dataÖR "qualityÖR "metrics AND software"

Um trotzdem die gewünsche Suchanfrage nutzen zu können, um potentiell noch spezifischere und relevantere Ergebnisse zu erhalten, nutzen wir zusätzlich die *Command Search* von IEEE Xplore, die auch verschachtelte boolsche Suchanfragen unterstützt [Iee].

Wir schauen uns jeweils die ersten 20 Egergebnisse an, da dies ein guter Kompromiss zwischen Relevanz und Anzahl der Ergebnisse ist.

Wir suchen nicht nur im Titel, sondern auch im Text, da es nicht immer alles expliziet im Title steht. Als Ziel des Papers, das ist zb. wichtig für die wörter quality oder metrics. Auch wenn viele das nicht explizit im Titel oder im abstract haben, wird trotzdem gibt es Paper diese daten als grundlage für die visualisierung verwendet, ohne dies wirklich explizit zu unterscheiden zwischen qualität und software selbst. ZB: Visualization of the Static Aspects of Software: A Survey

Außerdem suchen mit der normalen Google Suche nach existierenden Tools oder anderen Seiten, Ideen die genutzt werden können, um Software-Qualitätsmetriken zu visualisieren. Nutzen wir verschiedenste suchanfragen.

3. Inklusion und Exklusion Kriterien/Paper auswahl

Auf die Suchergebnisse wenden wir noch folgende Filter an, um unrelevante oder nicht zugängliche Ergebnisse zu exkludieren:

1. Wie bereits gesagt für google scholar: software ist ein muss, der rest ist optional in google scholar, auch wenn wir eigentlich gerne hätten, dass zumindest eines der wörter "qualityöder "metrics"vorkommt, um sicher zu gehen, dass es sich um Software-Qualitätsmetriken handelt und nicht um software-Visualisierung im allgemeinen (die wichtigkeit der unterschiedung wurde im Grundlagen Abschnitt 2.2 erläutert) deswegen filtern wir die suchergebnisse auch manuell, um sicher zu gehen, dass es sich um relevante Ergebnisse handelt: zB: Software Architecture Visualization: An Evaluation Framework and Its Application [GHMo8]

2. in englisch oder deutsch - aber alles war sowieso auf englisch, aufgrund der Suchbegriffe, außer ein spanisches paper: Software visualization tools and techniques: A systematic review of the literature[Cru+16]

3. Frei zugänglich über alle herkömmlichen Anbieter: IEEE, Springerlink, ACM digital library, ResearchGate,... Das sind die meisten außer z.B. Understanding software evolution using a combination of software visualization and software metrics [LDo2]

4. Außerdem generell innerhalb auch passen. z.B. Open source software for visualization and quality control of continuous hydrologic and water quality sensor data [Hor+15]

5. wird wirklich eine Visualisierung vorgestellt? Wenn in dem Paper entweder, mindestens eine 2D Layout vorgestellt wird oder wenn mindestens eine 2.5D Visualisierung vorgestellt wird. z.B.: Existing model metrics and relations to model quality [MDo9]

Für die normale Google Suche, verwenden wir verschiedene heuristische Filter um Werbung, Dienstleistungsanbieter und andere irrelevante Ergebnisse zu filtern.

4. Auswertung der Paper und Extraktion der Layouts

Filtern der Layouts, könnten diese für 2.5D geeignet sein? Sammeln und gruppieren der geeigneten verschiedenen Layouts Entscheiden für Layouts

Wie entscheide ich, ob ein Layout geeignet ist: - Es sollte eine Art von Metrik oder Wert darstellen können. - Es sollte eine Art von Layout haben, dass die Struktur der Hierarchie darstellt. - Es sollte ins drei dimensionale extrahierbar sein. - Es soll auch ohne spezielle Einfärbung im 2D funktionieren - eventuell nur die Ordner einfärben

5. Qualitätsanalyse: (Analyse der Layouts und Entscheidung welche Layouts in dieser Arbeit behandelt werden)

5.2 Durchführung der Suche

Hier stellen wir zunächst die Ergebnisse der Suche vor nachdem bereits die Filter angewendet wurden.

Erste Google Scholar Suche: 1. Voronoi treemaps for the visualization of software metrics [BDLo5] - Voronoi Treemap Layout - sie sagen, dass das Problem ist, dass Geschwister nicht einfach von anderen benachbarten Knoten unterschieden werden können, wir versuchen, dass mit den Margins zu lösen - als erste probieren sie nicht rechtecke sondern Polygone

The problem is provoked by the square-like shape of the rectangles, and because the edges are only horizontally and vertically aligned, whereby the edges of the different objects appear to run into each other. [...] A solution for this problem is the layout of Treemaps based on non-rectangular objects. [BDLo5, S. 3]

these polygons are clearly distinguishable because of their irregular shapes, and their aspect ratio converges to one [BDLo5, S. 6]

Aber selbst sie fügen kanten hinzu - und größere he höher außerdem nutzen sie Sättigung um noch mehr die Tiefe und die Hierarchie darzustellen - Kritik: sie wollen treemap geschwisterproblem lösen, schaffen dies aber durch ihren ansatz nicht wirklich und fügen dann doch wieder kanten und farbe hinzu um das zu unterstützen

2. Visualization-based analysis of quality for large-scale software systems [LSPo5] - Zeichnen auch Quader. diese Quader haben immer die selbe grundfläche - metriken werden dargestellt durch 1. die höhe des Quaders, 2. die farbe des Quaders, 3. die rotation (zwischen 0 und 90 Grad) - deren problem ist also nur die platzierung der Quader - KRitik: gefällt mir nicht so gut, vorallem die Roation, außerdem viel platz verschwendet. Sie sagen, dass gedreht schlecht ist. aber 90 Grad sieht nicht so schlimm aus wie 45 grad. dumm -Struktur auf klassen und paket ebene -> das macht es natürlich einfacher, da es generell wenier packete als ordner gibt (packete können auch nochmal in ordner unterteilt sein) - Am ende nutzen sie für die platzierung treemap und Sunburst - Da die fläche fix und diskret ist, erweitern sie sowohl den treemap als auch den sunburst algoritmus - Treemap: beim slicen, wenn der platz zu schmal wird, wird der platz einfach erweitert, für alle knoten, in eine richtung -> sehr viel leerer platz, auch wenn sie versuchen das so gering wie möglich zu halten "burst can still result in the insertion of holes in the representations. In simple illustrations such as in Figure 4, this looks like a serious problem. However, our framework is designed to study systems made of hundreds and thousands of elements. In fact, in the average cases (real softwares) that we tested, it did not prove a problem at all in our visualization." Zeigt, wie wichtig unsere datenanalyse ist, da es wichtig ist die visualisierung für reale daten zu machen und nicht nur simple beispiele vielleicht etwas zu simple gedacht??

"there is still room for improvements, mainly to better exploit the entire display space" [LSPo5, S. 8]

3. Visualization of the static aspects of software: A survey [CZ10] In diesem Paper werden verschiedene Visualisierungen vorgestellt. dieses paper auch schon in verwandten arbeiten erwähnt. Es werden 6 verschiedene Visualisierungen vorgestellt, die auch auf unseren usecase passen.

Sehr gute arbeit, da sie einige visualiasierungen vorstellen. 2D: Treemap, Sunburst, Circle Packing, voronoi

Verschiedene Stadtmetaphern Eine stadt und insel metaphor, solar system metaphor, welche gleich auch noch genauer beschrieben wird hierarchical net

Zeigen auch, dass das was am meisten verwendet ist stadt metaphor und generell treemaps

Ansonsten auch viele andere visualisierungen, die nicht so gut passen, weil es speziell um architektur, evolution oder verbindungen geht.

4. Visual realism for the visualization of software metrics [HVWVWo5] Nutzen normale Cushion Treemaps und schattierung und struktur um metrik

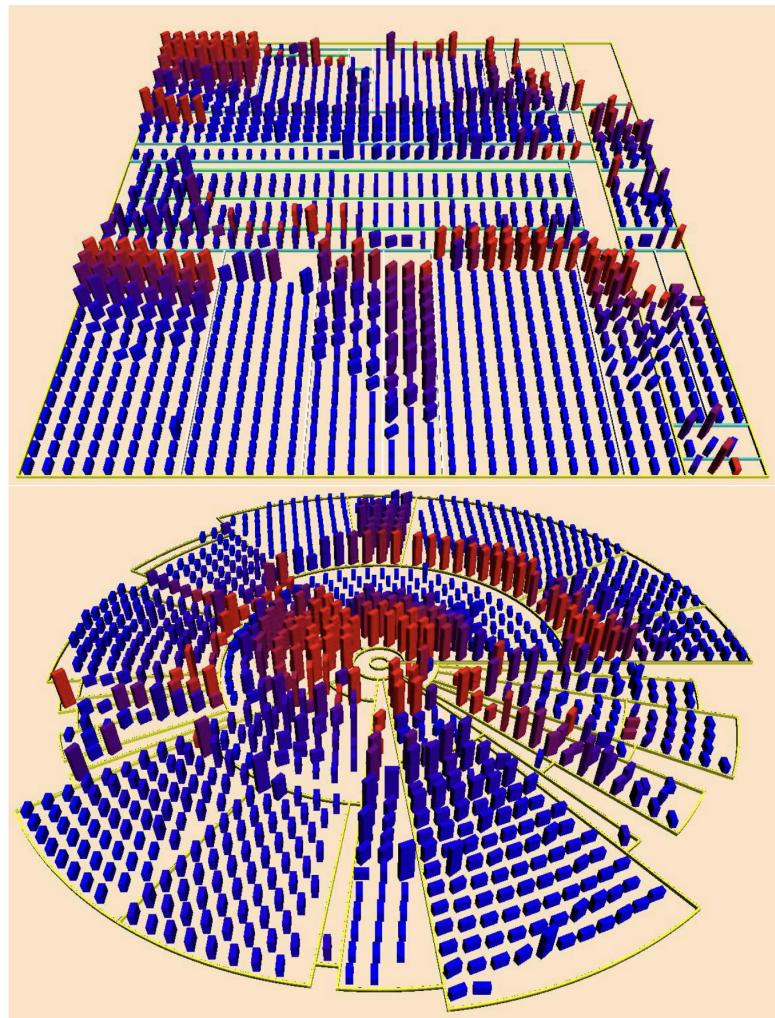


Abbildung 5.1: (Oben) Modifizierte Treemap-Technik und (unten) modifizierte Sunburst-Technik. Beide repräsentieren PCGEN, ein Tool zur Charaktererstellung in RPG (1129 Klassen). [LSPo5, S. 5].

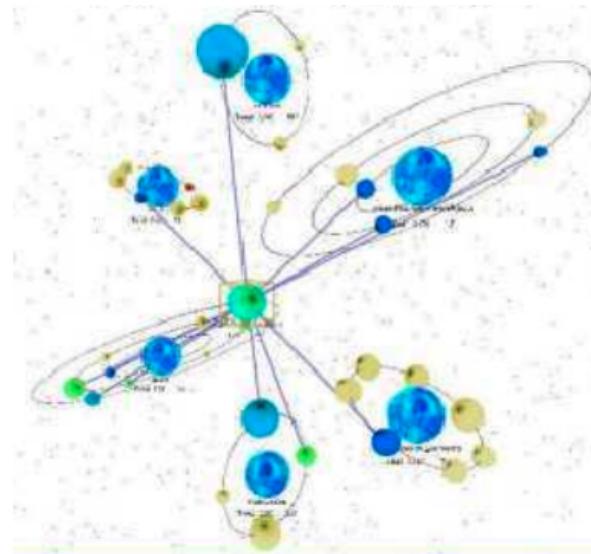


Abbildung 5.2: Solar System Metapher für 3D Visualisierung von objektorientierten Softwaremetriken. Die farben des bildes sind invertiert, um die lesbarkeit im ausdruck zu verbessern. [GYBo4, S. 4]

werte dar zu stellen nutzen aber auch package struktur, die deutlich einfacher ist als ordner struktur -> nur drei level

5. An overview of 3D software visualization [TCo8] Zeigen alles von 2D, 2.5D und 3D

6. CityVR: Gameful software visualization [Mer+17] - Stellen stadt metaphor in 3D vor - fokus auf gamification - layout ist ähnlich wie bei codecity

7. A solar system metaphor for 3D visualisation of object oriented software metrics [GYBo4] Schauen diese metriken an: Lines of Code(LOC) NCSS Number of Children(NOC) DF Weighted Methods (WMC) NCSS Coupling Between Objects(CBO) DF Depth of Inheritance Tree (DIT) DF ganz neue metaphor: solar system

Man erkennt, dass es wirklich 3D ist und nicht 2.5D Sie kritisieren, dass die Stadt-Metapher nicht so gut ist, da sie nicht wirklich 3D ist. Und keine Komplexen Zusammenhänge dargestellt werden können. Überdeckung.

Sonnen sind pakete. Planeten sind Klassen, wobei die farbe den typ der klasse angibt. Umlaufbahnen stellen die Vererbungsebenen innerhalb des Pakets dar. Planeten in derselben Umlaufbahn gehören zur gleichen Vererbungshierarchie Die Größe der Planeten ist eine Metrik (die anzahl der zeilen code) Verbindungen zeigen kopplung zwischen Klassen an.

Es werden aber nur relativ kleine Pakete visualisiert, mit nur so ca. 30-40 klassen.

es können nur 2 metriken dargestellt werden, eine auf klassenebene, eine auf verbindungsebene. Man könnte noch die Farbe als metrik nutzen und man könnte nicht auf klassen sondern auf file ebene betrachten (geht ja eigentlich immer) Es wird gefühlt sehr viel leerer platz verschwendet Es gibt doch auch einige überdeckungen, da die umlaufbahnen, teilweise hinter an-



Abbildung 5.3: Ein Beispiel für eine EvoSpaces Visualisierung. Die Gebäude repräsentieren Klassen, je nach Metrik-Wert werden diese in unterschiedliche Kategorien eingeteilt (z.B. bis zu 50 LOC - Wohnhaus, 51-200 LOC - Apartmentblock,...) [ADo7, S. 3].

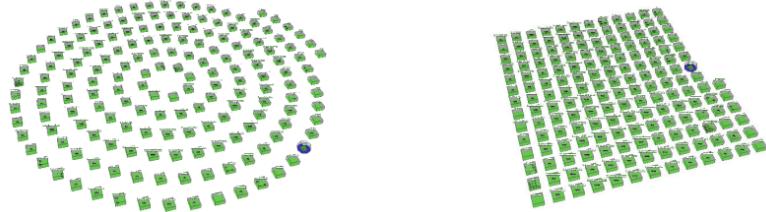


Abbildung 5.4: Die zwei Layouts, die EvoSpaces verwendet. Links: Konzentrisch und Rechts Schachbrettartig. [ADo7, S. 3].

deren planeten hergehen. Man könnte das ganze als 2D Layout verwenden und dann kugelförmig extrudieren, das ich stelle es mir unvorteilhaft vor

8. EvoSpaces: 3D Visualization of Software Architecture. [ADo7] Sie untersuchen verschiedene Darstellungen simple Darstellungen wie einfache Quadern oder auch wirkliche Häuser. Metriken werden Transparenz, Höhe, Größe, Textur,... dabei können mehrere Metriken verwendet werden. Gebäude sind Klassen oder Dateien. Verbindungen sind Röhren bzw. Strahle die oben aus einem Gebäude herauskommen und dann im Bogen in ein anderes Gebäude von oben hineingehen.

sie stellen aber nur zwei Layouts vor, die nicht die Struktur zeigen, sondern auch wieder eine Metrik darstellen. z.B. im Kreis und älteste Datei am weitesten innen.

9. 3D representations for software visualization [MFM03] Weiterentwicklung von 2D Pixel Bars ins 3D, weil mehr Metriken und besser die Verbindungen darstellbar. softvis3d läuft auch auf Basis von Metriken auf Datei-Ebene sehr detailliert, viel kleineteiliger Darstellung, da einzelne Code-Zeilen dargestellt werden. Vor allem für die Entwicklung gedacht. Überdeckung wollen sie mit Transparenz lösen. Layout angeordnet nach Zeile für Zeile keine Hierarchie.

10. Software cartography: Thematic software visualization with consistent layout [Kuh+10] Es wird eine topographische Kartenmetapher verwendet. Dateien sind Hügel, die Höhe ist eine Metrik (z.B. LOC). Das Layout wird mit Hilfe eines speziellen Multidimensionale Skalierungsalgorithmus (Hit-MDS) erstellt. Als Stress Wert der Funktion wird die Lexikalische Distanz zwischen den Dateinamen verwendet (ist vielleicht gut, wenn man schnell bestimmte Dateien finden will, aber nicht so gut für die Hierarchie und auch weit weg).

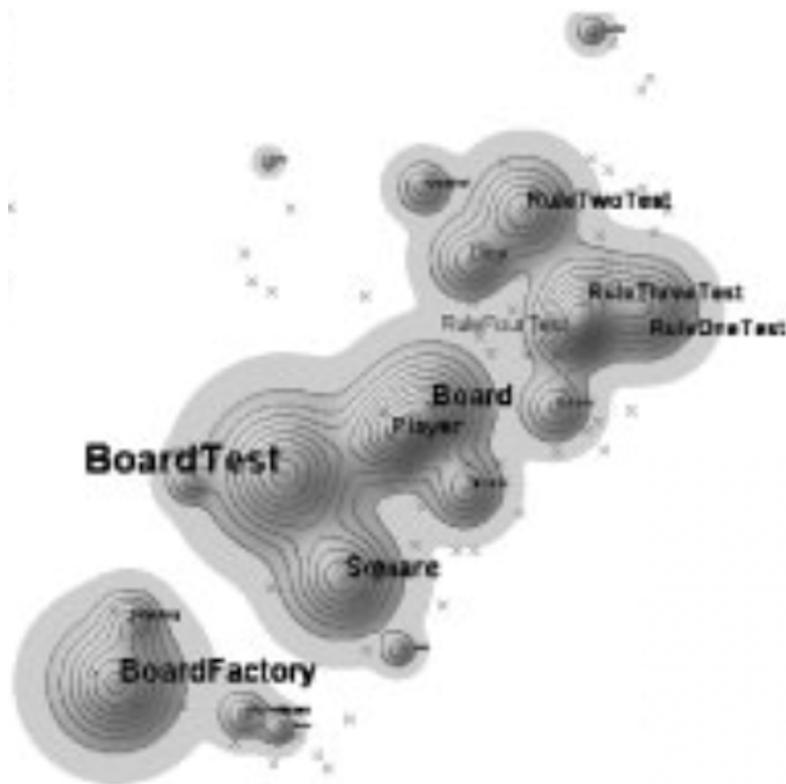


Abbildung 5.5: Software Cartography: Topographische Kartenmetapher für Software Visualisierung. Die Hügel repräsentieren Dateien, die Höhe ist eine Metrik (z.B. LOC). [Kuh+10].

von der software selbst) sie zeigen noch eine erweiterung mit icons auf den bergspriten, die auch noch eine metrik darstellen können wurde besonders entwickelt mit dem Ziel von software evolution

ich sage, das ist eine interessante grundidee und anders

zweite Google scholar suche: Doppelte Treffer: 4 zB.: Visualization-based analysis of quality for large-scale software systems [LSP05] 1. Exploring Relations within Software Systems Using Treemap Enhanced Hierarchical Graphs [BD05] auf oberer Ebene wird energy model of Noack and Lewerentz genutzt um einen graphen zu malen. dieser graph kann auch mehrere hierarchie ebenen visualisieren, dabei wird farbe als unterstützendes mittel genutzt, um die verschiedenen pakete oder order zu unterscheiden. Jeder knoten hat dann nochmal intern eine Treemap, die die Struktur des knotens zeigt

interaktion mit zoomen ist von nötien. Alles bisher noch in 2D

2. Visualizing Software Metrics in a Software System Hierarchy [Bur15] im grunde eine link treediagram, Metrik stripes - metrische streifen, zeigen über breite und farbe verschiedene metriken an

layout wird wie in groundlagen gesagt, sehr schnell unübersichtlich.

3. A Stable Greedy Insertion Treemap Algorithm for Software Evolution Visualization [VCT18] stellen nur neuen treemap algorithmus vor, der fokus auf stabilität hat diese wurden erstmals hier vorgestellt [Gor+18]

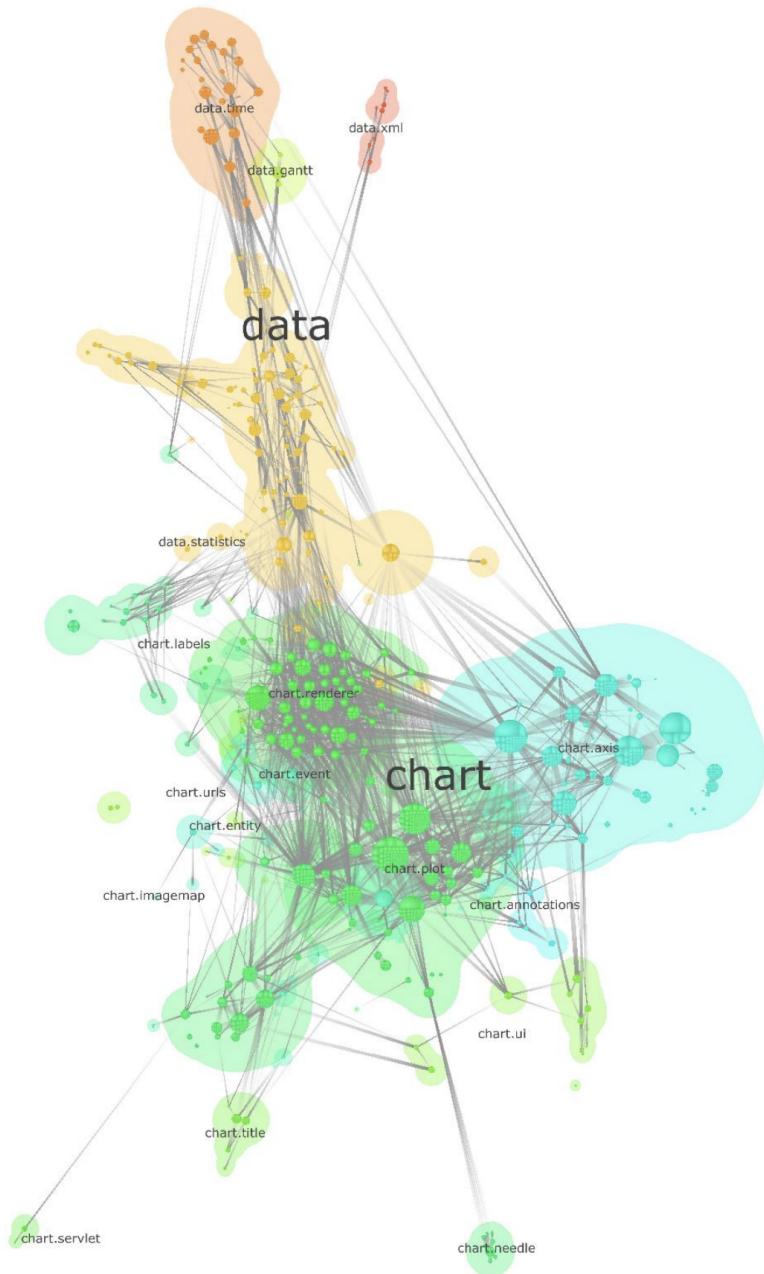


Abbildung 5.6: Exploring Relations within Software Systems Using Treemap Enhanced Hierarchical Graphs. Die obere Ebene ist ein Graph, der die Beziehungen zwischen den Paketen zeigt. Die untere Ebene (in den einzelnen Kreisen nur klein zu sehen) ist eine Treemap, die die Struktur des Pakets zeigt. [BD05, S. 5].

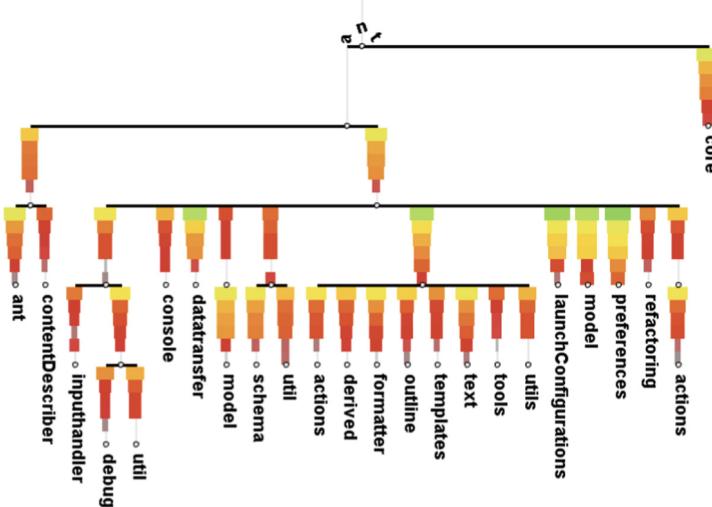


Abbildung 5.7: Im grunde ein link-treediagramm, das mit *Metric Stripes* verschiedene metriken darstellt. [Bur15, S. 740].

4. Visualizing Program Quality – A Topological Taxonomy of Features [AO+19] eigentlich gar nicht für software qualität, aber schlagen die bubble treemap vor

5. Stable and predictable Voronoi treemaps for software quality monitoring [HH17] erweitern voronoi 3 metriken: 1. farbe 2. größe 3. reihenfolge der anordnung/bzw positionierung idee ist als 3. metrk den fully qualified name zu nutzen um so die topologie der hierarchie darzustellen -> Ziel vergleich zwischen verschiedenen versionen

6. Visualization and evolution of software architectures [Kha+12]

IEEE Suche: Doppelt: 3 zB. Visualization of the Static Aspects of Software: A Survey [CZ10] 1. Metrics-based 3D visualization of large object-oriented programs [LS02] auch größe und farbe als metrik

Einzelne Kugeln, Würfel,... im 3d raum, die mit linien verbunden sind, je nach verbindung das layout ist nach einem gleichheitswert der klassen angeordnet, dieser wert kann alles sein, entweder metrik oder platz etc.

2. Visualizing Metric Trends for Software Portfolio Quality Management [Gen+21] Stellen einfach eine Plattform vor, die verschiedene Metriken über zeit visualisiert mit linien plots und balken diagrammen für komplette software - nicht wirklich relevant für uns

3. E-Quality: A graph based object oriented software quality visualization tool [ETB11]

First, we examine a small-sized project to be able to easily illustrate the features of E-Quality. [ETB11, S. 5]

nur vergleich mit folder based navigations methode

packete werden kreisförmig angeordnet Metriken haben auswirkung auf, muster, größe, farbe, form

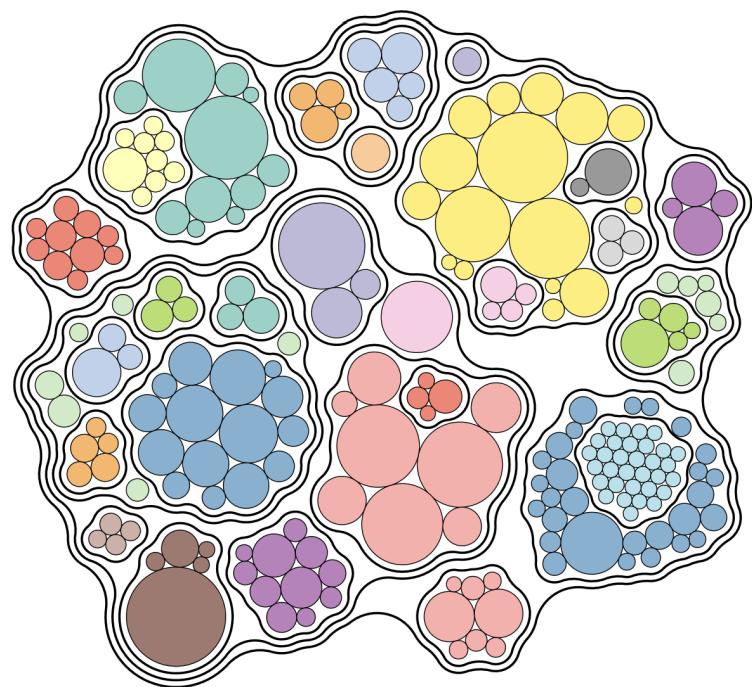


Abbildung 5.8: Beispiel für eine Bubble Treemap [Gor+18, S. 7]

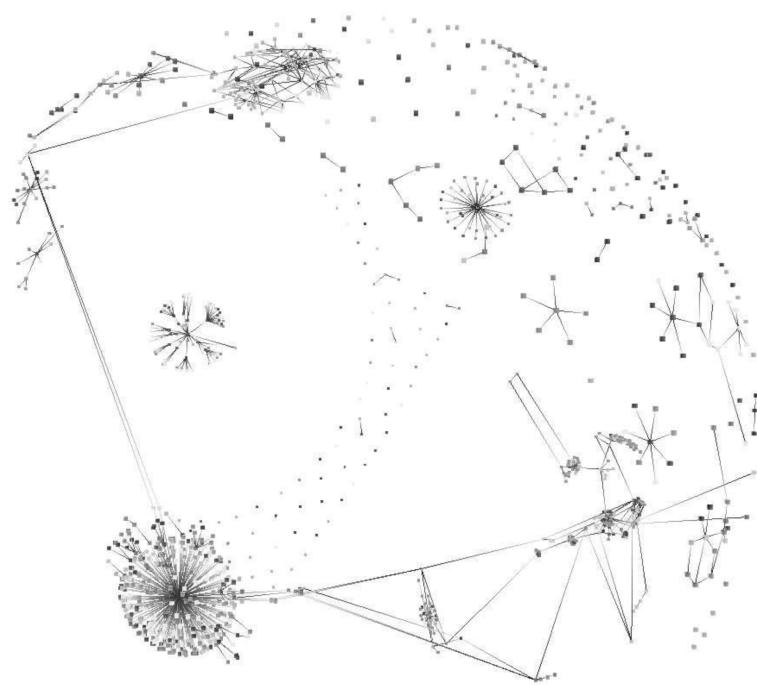


Abbildung 5.9: Die vorgestellte visualierungen zeigt, wie klassen in 3D gemappt werden anahdn von similarity score [LSo2, S. 5].

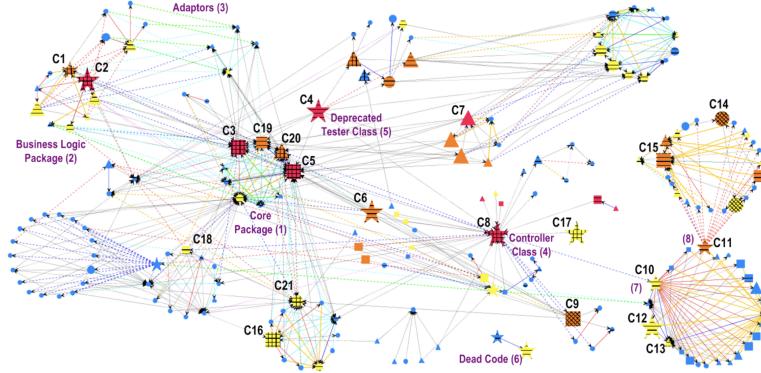


Abbildung 5.10: E-Quality: Ein graph basiertes objektorientiertes Software-Qualitätsvisualisierungstool. [ETB11, S. 5].

4. Visualization of Software Quality Expert Assessment [HB19] auf projekt ebene

5. QScored: An Open Platform for Code Quality Ranking and Visualization [TKS20] stellen eigentlich nur plattform vor visualisieren mit treemaps und sunburst und dependecy graph

6. Using the City Metaphor for Visualizing Test-Related Metrics [Bal+16] Stadt methapher in minecraft Gebäude sind sehr detailliert und bilden methoden, attribute, metriken, interfaces, ... alles mögliche ab eher eine spielerei, aber im grunde layout wie code city

7. Visual Analytics of Software Structure and Metrics [Kha+15] Nutzen den city view: von ecity [Kha+13] eCITY: A Tool to Track Software Structural Changes Using an Evolving City

Google Tool suche: Gefiltert werden paper, wenn sie kommen. Nur webseiten werden geöffnet Software Quality Visualizer

Software Metric Visualizer/software quality analysis/ software visualisation tool/ dabei auch die filter: 1. software quality vis 2. existing tool 3. keine paper, viele dienstleister raus filtern wie zb. <https://bitsea.de/dienstleistungen/software-qualitaetsanalyse/> -> nur dienstleistungen anbieten dabei gestoßen auf: <https://home.uni-leipzig.de/svis/publications.html> und <https://hpi.de/doellner/publications.html>

grafana.com -> aussortiert, weil generelle library für visualisierungen und dashboards

<https://github.com/MaibornWolff/codecharta>

www.jarchitect.com und [sonarqube](http://sonarqube.org) <https://www.cs.rug.nl/svcg/SoftVis/ArchiVis>

GETAVIZ gefunden sereen gefunden

code-is-beautiful: letzter commit vor 7 Jahren <https://github.com/quantifiedcode/code-is-beautiful> bietet: code city, Sunburst (2d): im grunde ein kreis diagram, welches von innen nach außen die ordner strukturen zeigt. die es können die größe (wie viel des kreises nimmt etwas ein) und die farbe als metric gewählt werden Stack (2d): im grund wie das kreis diagram nur von oben nach unten

Code Radar: <https://github.com/pschild/CodeRadarVisualization> Wie code city speziell auf vergleich von versionen ausgelegt, indem man zwei karten direkt nebeneinander sehen kann letzter commit vor 7 jahren

softvis: vor 2 jahren wie code city

Ndepend: <https://www.ndepend.com/docs/treemap-visualization-of-code-metrics?cx=0150956779873219162952D> Aber interessant: hier werden die kanten dunkel eingefärbt, um die einzelnen elemente visuell zu unterscheiden

<https://community.sap.com/t5/welcome-corner-blog-posts/i-have-a-dream-code-visualization/bc-p/13485189/highlight/true>: hat die idee dass gebäude wirklich echt sein könnten ein gebäude an dem gebaut wird, wurde kürzlich geändert alte gebäude wurden lange nicht mehr verändert oft geänderte objekte sind nah an einem hafen oder so

ein kommentar auf deren seite: "Playing around with these metrics seems like an easy way to identify those objects where a refactoring promises to be most beneficial, because large and complex objects that are changed often tend to introduce bugs and slow down the development process.

It also seems that this visual approach helps to promote topics like clean code within an organisation as the negative impact of such skyscraper classes becomes clearer when you look at these graphics.

I think I will take a Code City Snapshot from time to time to track how the red skyscrapers steadily turn into beautiful, clean and green suburbs.

Even without medieval buildings that can be accessed using VR, this might help a lot to navigate towards a clean code base."

<https://home.uni-leipzig.de/svis/getaviz/index.php?setup=web/Cityauchsehr sehr nice mit bausteinen verschiedene sachen visualisieren>

<https://codescene.com/product: 2d kreis diagramme>

<https://github.com/adamtornhill/code-maat> bietet auch einiges in 2d

Wie ist der Stand der Forschung? overview of 3d software visualisierung:

<https://ieeexplore.ieee.org/document/4564449>

interessant: <https://opus-hwtw-aalen.bsz-bw.de/frontdoor/deliver/index/docId/658/file/ICCSI SEE.pdf>

[Mer+18]: A systematic literature review of software visualization evaluation: Wie kann man visualisierungen bewerten? Hier geht es vor allem darum: "help analysts make sense of multivariate data (Merino et al., 2015), to support programmers in comprehending the architecture of systems (Panas et al., 2016), to help researchers analyze version control repositories (Greene et al., 2017), and to aid developers of software product lines"

Quality models are usually defined based on concrete measurements of software metrics (N. Fenton and J. Bieman, Software metrics: a rigorous and practical approach. CRC Press, 2014)

4 metriken werden visualisiert: farbe, position, höhe, breite

sehr interessant. 1. Idee unterschiedliche metriken zu stacken 2. gibt einen generelen qualitäts wert am ende heraus, erstellt aber eine grafik, die die einflüsse verschiedener klassen auf die verschiedenen metriken zeigt und die generelle bedeutung für die allgemein metrik am ende.

Moose - ich würde das eher bewerten für entwickler. Es werden hier verbindungen von klassen dargestellt - abhängigkeiten undco.

VON cascaded: hier könnte man auch nochmal suchen, muss aber auch nicht. So etwas ähnliches auch sagen: A number of other hierarchy visualization techniques have been developed [18, 23, 29, 32], including space-filling visualizations like step trees [6], Voronoi treemaps [2] and generalized treemaps [34]. Although relevant to hierarchy visualization, we pursue contributions that are sufficiently distinct from such work that we do not dwell on extensive comparisons. [LFo8]

5.3 *Dokumentation der Ergebnisse und Vorstellung der Layout-Umsetzung*

In diesem Abschnitt stellen wir die Ergebnisse der Recherche vor und dokumentieren die Layouts, die wir in dieser Arbeit umsetzen wollen.

5.3.1 *Sunburst Layout*

5.3.2 *Circle Packing Layout*

5.3.3 *Noch eins mehr?*

6

DATEN ANALYSE

Dieser Abschnitt hat 2 Ziele 1. eine gute Auswahl an Codebasen finden, die wir sowohl für unsere Algorithmus Anpassungen als auch für die Evaluation am Ende verwenden können. 2. wollen wir die Struktur von Softwareprojekten näher untersuchen, um zu verstehen, wie sie aufgebaut sind und welche Eigenschaften sie haben. Wie gesagt wir wollen die Struktur näher bestimmen. Wie definiert ist die Struktur immer eine Baumstruktur. Genauso wie in der Problemstellung (Abschnitt 3) definiert.

6.1 Auswahl der Projekte

Als Quelle werden wir GitHub verwenden, da es eine große Anzahl an Open-Source-Projekten bietet und die Daten leicht zugänglich sind. Wir gehen nicht davon aus, dass es einen signifikanten Unterschied zwischen Open-Source und Closed-Source Projekten gibt [Rag+05; PSEo4], weshalb wir nur auf open source zugreifen. Wir probieren möglichst breit gefächert Projekte zu betrachten, um ein möglichst breites Spektrum an Softwareprojekten abzudecken. Das Ziel bei der Auswahl der Projekte ist es, eine möglichst diverse Auswahl an Projekten zu haben, um später eine Aussage über die Struktur von Softwareprojekten im Allgemeinen treffen zu können und nicht nur über die Struktur der meisten Softwareprojekte. Wir könnten einfach zufällige Projekte von Github auswählen, aber das würde vermutlich zu einer Verzerrung führen und eher kleinere und unbekannte Projekte auswählen. Deshalb suchen wir nach Kriterien, die einen Einfluss auf die Struktur von Softwareprojekten haben könnten.

da es keine Quellen gibt, die eine solche Auswahl an Kriterien vorgeben, können wir nur Vermutungen anstellen und diese im Nachhinein anhand der Daten überprüfen, können aber nicht sicher garantieren, dass wir wirklich alle relevanten Kriterien berücksichtigt haben. Aber das ist ja auch keine Arbeit über die Selektion von Projekten, deshalb geben wir uns damit zufrieden. Zur Identifikation von relevanten Kriterien richten wir uns nach den Informationen, die die GitHub API uns zur Verfügung stellt. Es gibt eine Vielzahl an Metainformationen.

Folgend checken wir erstmal die Kriterien, ob diese überhaupt relevant sind und ob sie einen Einfluss auf die Struktur von Softwareprojekten haben.

Programmiersprache

Wir werden Projekte in verschiedenen Programmiersprachen betrachten, da die Struktur von Softwareprojekten in verschiedenen Programmiersprachen stark unterschiedlich sein kann - hier fehlt noch ein Beweis - ist da überhaupt

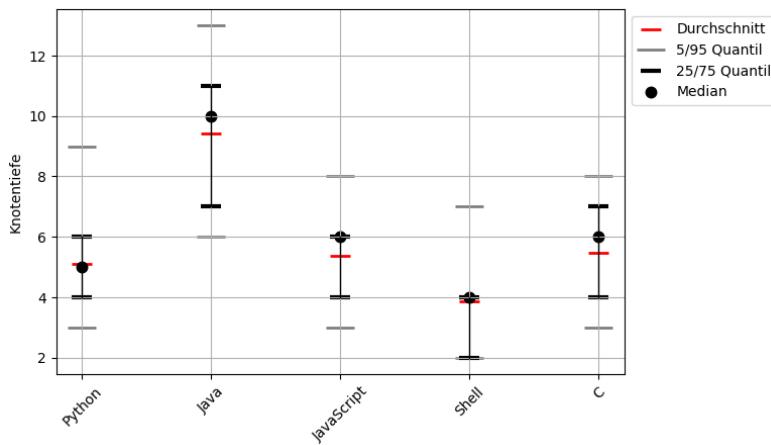


Abbildung 6.1: Die Tiefe variiert stark nach Sprache. Zum Beispiel sind die meisten Knoten (95%) in java deutlich tiefer (tief 7) gelgen als die meisten Knoten (75%) in shell (tief 4)

so? wenn nicht kann man das auch weglassen. Wir wählen die Programmiersprachen nach ihrer Popularität und Paradigma aus.

- **Python:** Die zweit populärste Sprache [Sofb] mit dynamischer Typisierung, sowohl imperativem als auch objektorientiertem Paradigma.
- **JavaScript:** die populärste Sprache [Sofb] mit dynamischer Typisierung. Die Struktur von JavaScript-Projekten ist besonders geprägt durch Frameworks.
- **Java:** Die dritt populärste Sprache [Sofb] mit statischer Typisierung und objektorientiertem Paradigma. Java-Projekte sind oft sehr strukturiert und folgen bestimmten Konventionen.
- **C:** Eine der ältesten Programmiersprachen, die immer noch weit verbreitet ist (Platz 9 [Sofb]). C-Projekte sind oft sehr nah an der Hardware und haben eine andere Struktur als Projekte in höheren Programmiersprachen.
- **Shell:** Eine Skriptsprache (Platz 8 [Sofb]), die oft für Automatisierung und Systemadministration verwendet wird.

Man sieht aber in beiden GRafiekn, dass die Sprache einen Einfluss auf die Struktur hat, was alles war was wir hiermit zeigen wollten.

Projektgröße

Die Projektgröße ist offensichtlich der Wichtigste Faktor für die STruktur eines Projekts. Wir gehen davon aus, dass Große Projekte auch dann größere Strukturen (also mehr knoten) haben (stimmt nicht zwingen - wie im folgenden Abschnitt 6.1.1 gezeigt wird) Github bietet leider nur die möglichkeit nach größe und nicht nach Anzahl der Dateien zu filtern, was uns eigentlich interessieren würde.

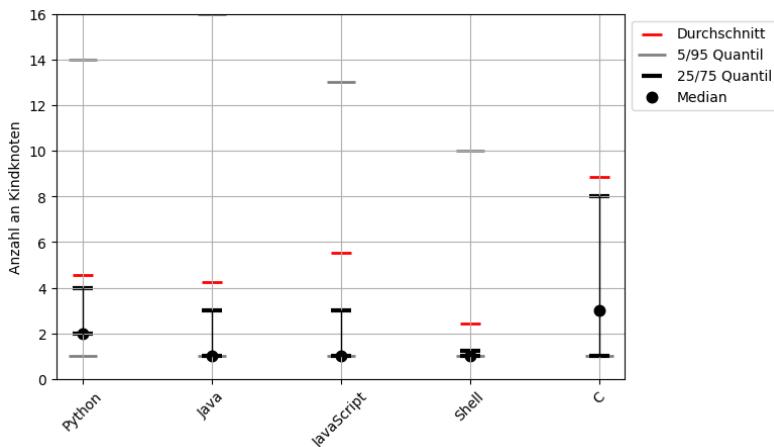


Abbildung 6.2: Die Varieiert zwar im Durchschnitt teilweise zB zwischen sheell und C aber im großen und ganzen sind die Intervalle der Quantile schon überlappend

Wir beweisen kurz, dass die Größe einen Einfluss auf die Struktur hat, indem die Hierarchie-Tiefe in Abhängigkeit von der Größe untersuchen.

Man sieht aber in beiden Grafiekn, dass die Größe einen Einfluss auf die Struktur hat, was alles war was wir hiermit zeigen wollten.

Ungenutzte Kategorien

Man könnte denken, dass auch andere Kategorien wie die Anzahl der Sterne oder die Anzahl der Mitwirkenden eines Projekts einen Einfluss auf die Struktur eines Projekts haben. Wir hatten auch diese vermutung und haben dies nachgeprüft. Um fragen zu vermeiden zeigen wir hier auch die ergebnisse, die gezeigt haben dass diese Kategorien keinen Einfluss haben und wir sie deshalb verworfen haben.

1. Wir gingen davon aus, dass die Popularität eines Projekts einen Einfluss auf die Qualität eines Projekts hat. Je Popularitärer ein Projekt ist, desto "besser" muss es sein, desto besser muss ja auch die Qualität sein. Und wir gehen davon aus, dass die Qualität eines Projekts einen Einfluss auf die Struktur des Projekts hat. Allerdings deutet eine Studie an, dass Popularität nicht in Korrelation mit Software Qualität steht [Saj+14]. Außerdem sind die meisten Projekte die auf GitHub viele Sterne haben eher Sammlungen von Büchern, Kursen oder anderen Ressourcen und nicht wirklich Softwareprojekte [evanli_github-ranking_2025], weshalb wir uns entschieden haben, die Popularität nicht als Kriterium zu verwenden, auch wenn es auf den ersten Blick sinnvoll erscheinen mag.

2. Eine größere Anzahl an Mitwirkenden, mehr Änderungen und generell mehr Aufmerksamkeit. Es gibt paper, die sagen, dass unterschiedlichen Anzahlen an Mitwirkenden hat einen Einfluss auf die Struktur und Qualität des Codes haben kann [CAC20]. Wir werden nur Projekte bereits ab einer Anzahl von 2 Mitwirkenden betrachten, da Projekte mit nur einem Mitwirkenden oft nicht wirklich repräsentativ für die Struktur von Softwa-

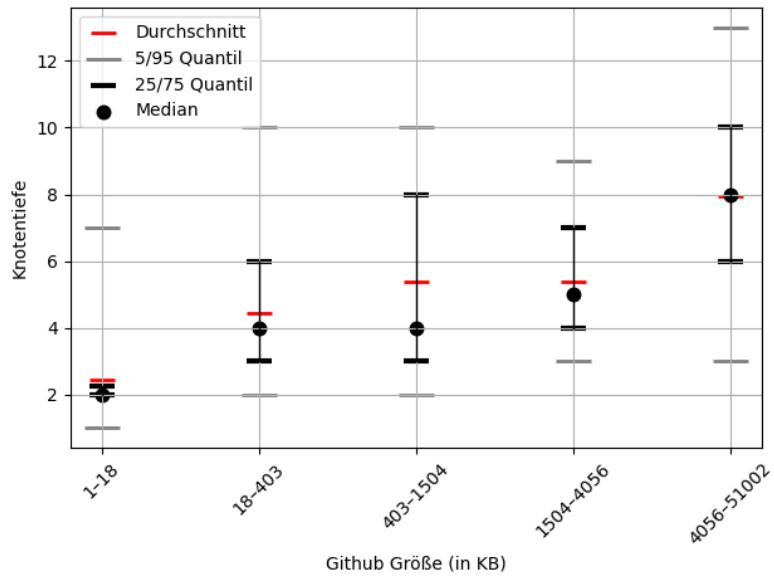


Abbildung 6.3: Die Hierarchie-Tiefe korreliert positiv mit der Größe des Repositories, was zeigt, dass größere Repositories tendenziell eine tiefere Hierarchie haben. Pearson Korrelation von 0.37, zeigt dass die Korrelation schon schwach ist, bzw es viele Ausreißer oder andere Faktoren gibt (wie zum Beispiel die Sprache)

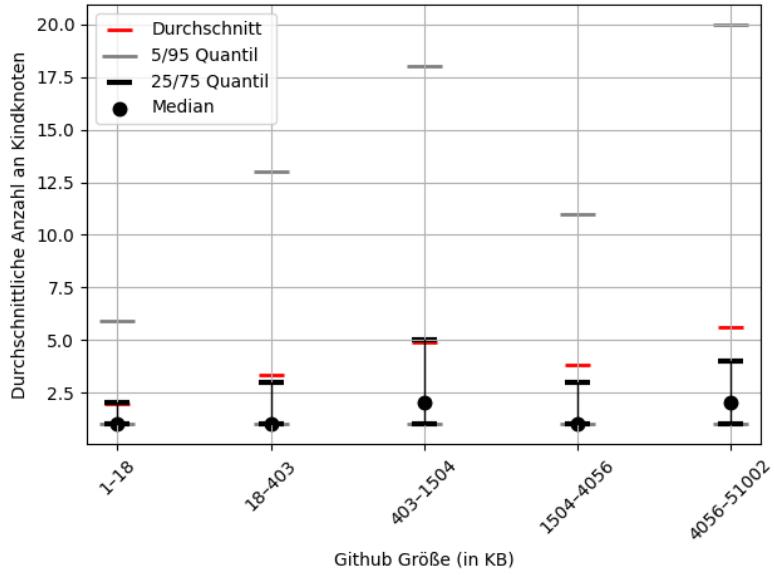


Abbildung 6.4: Die Durchschnittliche Anzahl an Kinder-Knoten allerdings nur leicht positiv mit der Größe des Repositories korreliert, was zeigt, dass größere Repositories tendenziell eine größere Anzahl an Kinder-Knoten haben, aber nicht wirklich ausschlaggebend. (Pearson Korrelation: 0.09)

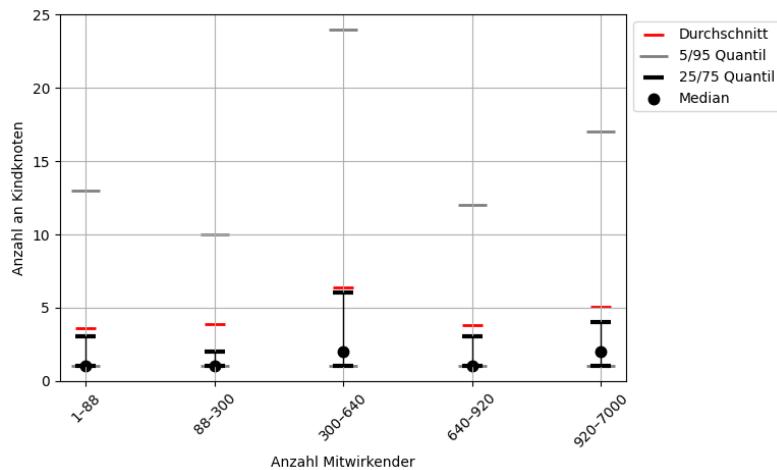


Abbildung 6.5: Die Anzahl der Mitwirkenden auf der x-Achse hat keinen erkennbaren Einfluss auf die Anzahl der Kinder-Knoten eines Knotens.

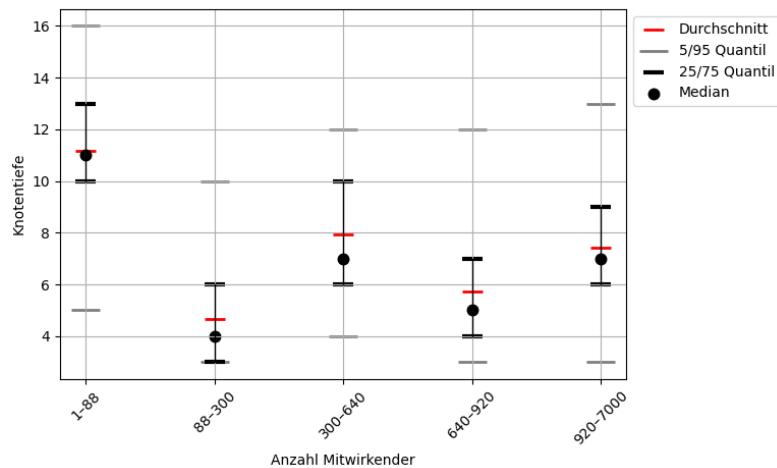


Abbildung 6.6: Die Anzahl der Mitwirkenden auf der x-Achse hat keinen erkennbaren Einfluss auf die Hierarchie-Tiefe eines Knotens.

reprojekten sind. Wir überprüfen, ob die Anzahl der Mitwirkenden einen Einfluss auf die Struktur - konkret die Hierarchie-Tiefe und die Anzahl der Kinder-Knoten - hat.

6.1.1 *Die Auswahl der Projekte im Detail*

So wir wissen jetzt, dass wir nur auf sprache und größe achten wollen. wir haben 5 verschiedene bekannte sprachen und verschiedene größen. Wir nehmen eine gute anzahl an Projekten, so dass jede kombi repräsentiert ist, aber trotzdem nur so viele wie wir rechnerisch handeln können. In Abbildung 6.7 haben wir die Verteilung der Projektgrößen von 2500 zufällig ausgewählten GitHub-Repositories dargestellt. Es ist zu erkennen, dass es deutlich mehr kleinere Projekte gibt. (Zur Größen einordnung: das REact repository, was als ein sehr großes Projekt gilt, hat eine Größe von ca. einem Gigabyte) Wir wollen die Größe von 98 Prozent aller Repositories abdecken. Wir haben uns

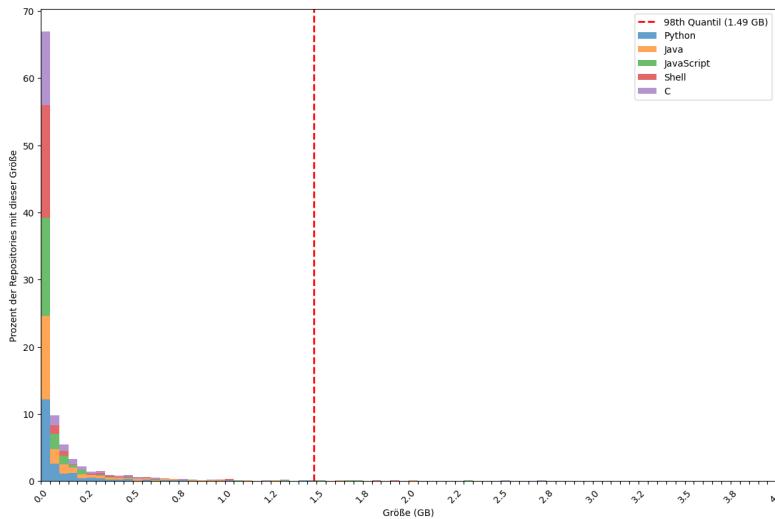


Abbildung 6.7: Das Diagramm zeigt, dass es deutlich mehr kleine Projekte gibt (logarithmisch abnehmend)

für diesen Wert entschieden, da wir eine sehr gute Abdeckung auch großer Projekte haben wollen, aber trotzdem auch nicht zu viele Projekte generell haben wollen. Außerdem ab dieser Größe wird es wahrscheinlich generell sehr schwer sinnvoll zu visualisieren bereits das React Repo kann mit 1GB teilweise unübersichtlich werden, nicht weil der algorithmus oder die visualisierung schlcht ist sondern weil es einfach zu viele Dateien werden irgendwann.

Man könnte argumentieren, warum macht ihr euch die arbeit und nehmt nicht einfach random Projekte? Weil wir spezifisch sicher gehen wollen projekte aller größen in einer linearen verteilung. ansonsten haben wir vor allem kleine Projekte, bei denen sowieso das treemap problem nicht so sehr auftritt, dann würden wir in den daten auch keine wirklichen änderungen sehen können (wahrscheinlich) oder nicht so große. weil eher kleine projekte ??.

Deswegen wählen wir spezifisch Projekte bis zu 1,5GB (98er Quantil) aus. Wir wählen das so, dass wir eine lineare verteilung haben und ca. alle 0,1GB ein Projekt - also haben wir 15 Projekte pro Sprache -> 70 Projekte, die die grundlage für alle analysen und evals dieser ARbeit bieten. Zusätzlich gehen wir sicher, dass Projekte wie *The most comprehensive database of Chinese poetry* oder *A list of funny and tricky JavaScript examples* nicht im datensatz enthalten sind, weil das keine echten Software-Applikationen sind, sondern sammlungen oder andere sachen, für die unsere Visualisierung nicht gedacht ist -> klar auch möglich natürlich, aber nich das Ziel dieser Arbeit.

In Abbildung 6.8 ist zu erkennen, dass wir relativ gleichmäßig die repos bis zu einer Größe von 1,5GB abdecken. Die genau Auswahl an repos ist in Anhang A.3 zu finden.

Da die größe nicht 1 zu 1 mit der Anzahl an nodes korreliert. ein order zum beispiel nimmt fast keine größe ein. außerdem sind bei jeder sprache die dateien verschieden groß es gibt unterschiedliche anzahl an dateien die

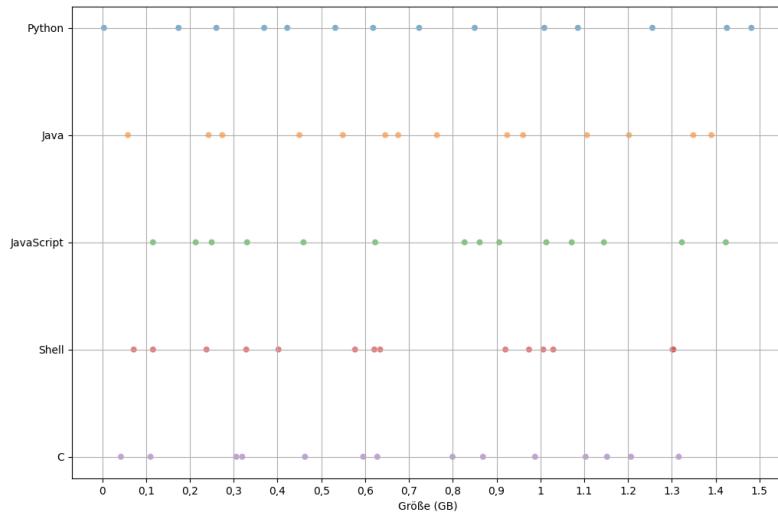


Abbildung 6.8: Die Verteilung der Projektgrößen der ausgewählten Repositories je Sprache.

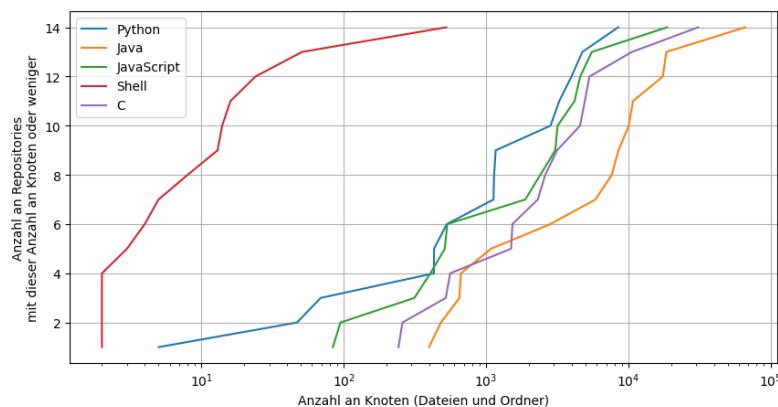


Abbildung 6.9: Die Verteilung der Anzahl an Knoten der ausgewählten Repositories je Sprache.

nicht betrachtet werden wie png, mp4 etc die zur Größe aber nicht zur Anzahl an nodes beitragen. Es ist zu erkennen, dass dadurch doch wieder mehr kleinenere Repos gibt, aber das akzeptieren wir einfach. 6.9 Besonders bei Shell ist zu erkennen, dass die Anzahl an Knoten nicht wirklich linear verteilt ist. Zeigt wie unterschiedlich die repos der verschiedenen Sprachen sind, trotzdem akzeptieren wir das, am Ende bildet es die Realität ab.

6.2 Struktur von Softwareprojekten

Wie berechnen wir die Struktur der Repositories und wie bekommen wir Metrik Daten für diese REpos? Wir clonen jedes repo und analysieren die Struktur der Repositories mit dem CodeCharta Analysis CLI Tools [Gmbd]. Dieses Tool analysiert die Struktur der Repositories und berechnet für jede Datei (es werden nur Dateien betrachtet die zur entsprechenden Programmiersprache gehören, also zB. .py für Python) die Metriken, die wir später

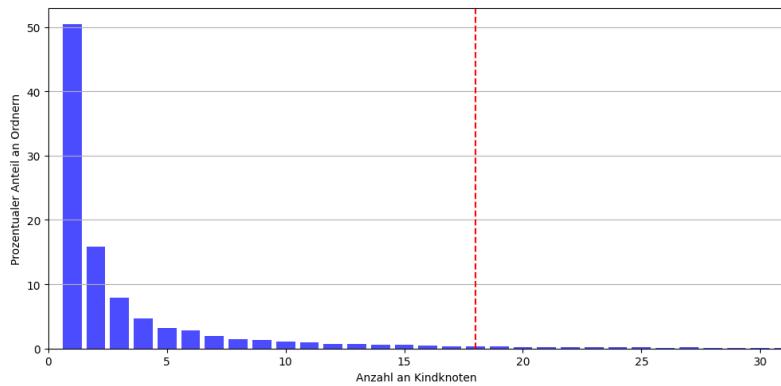


Abbildung 6.10: Die Anzahl der Kind-Knoten pro Ordner. Mehr als 50% der Ordner haben nur ein Kind. Das 95 Prozent Quantil liegt bei 18.

für die Visualisierung verwenden werden. Genutzt wird der unified parser mit diesem Command: Als ergebnis bekommen wir eine JSON-Datei, mit der in Abschnitt 3 definierten Struktur. Ein Beispiel ist auch im anhang (Listing A.1) zu finden.

Im Abschnitt Problemstellung (Abschnitt 3) wurde das Format der Eingabe-Daten formal definiert (siehe Listing 3.1). Die Hierarschische-Struktur selbst ist offen in ihrer Größe und Form also zB. wie viele Kinder hat ein Knoten, wie Tief ist die Verschachtelung etc. Wir versuchen das in diesem Kapitel einzuschreänken, also auf zu zeigen, wie hierarschiche strukturen, die aus software analyse auf file-basis laufen aussehen. UM nochmal klarer zu werden: die formale definition steht. aber in der praxis realität werden diese Strukturen aus der Datei-Hierachie von Software-Projekten entstehen.

Warum wollen wir das tun? Weil wir uns erhoffen, erkenntnis über die Datei-Struktur von Software-Projekten zu gewinnen, die uns helfen, die algorithmen genau für diese Strukturen zu optimieren und außerdem die Laufzeiten der algorithmen eine bessere einschätzung zu geben.

Im Grundlagen Kapitel wurde bereits erwähnt, dass die Metrik-Daten, die wir durch die Analyse von Code erhalten, spezielle Eigenschaften aufweist (siehe Abschnitt 2.1). Diese Eigenschaften sind wichtig, um die Visualierung zu optimieren und zu wissen in welchem Rahmen sich die zu visualisierenden Daten bewegen.

numberOfFile: 3262.34 (avr), 708.50 (median), 7391.42 (std)
 numberofFolders: 958.76 (avr), 242.00 (median), 2089.59 (std)
 numberofNodes: 4221.10 (avr), 1108.50 (median), 9308.05 (std)

sehr hohe varianz zwischen den einzelnen repos.

in abbildung 6.10 ist zu erkennen, dass die meisten Ordner nur ein Kind haben (was auffällig ist, sie Ordnerketten im folgenden). Das quantil 95 liegt bei 18, was interessant wird im Abschnitt

6.2.1 Orderketten

Wir vermuten, dass die hohe Tiefe bei java daher kommt, dass es in Java üblich ist, die Struktur des Codes in Paketen zu organisieren, die häufig

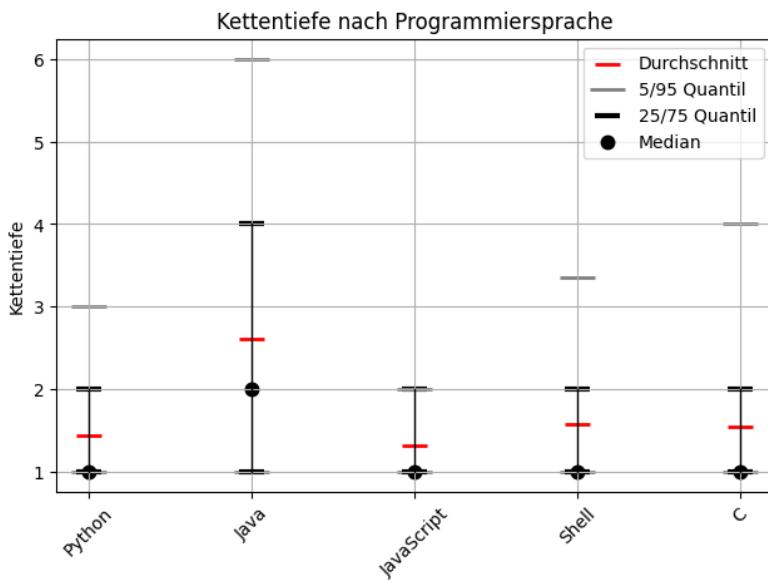


Abbildung 6.11: Die Kettentiefe ist besonders bei Java auffällig variabel und auch der Mittelwert ist höher als der bei allen anderen Sprachen.

in mehreren Ebenen verschachtelt sind. Zb: Zum Beispiel im OpenSource Projekt CodeCharta [Gmbb] `codecharta/analysis/analysers/AnalyserInterface/src/main/kotlin/de/maibornwolff/codecharta/analysers/analyserinterface/AnalyserInterface`. Ab dem Ordner `main` bis zum Ordner `analyserinterface` geht die Kette von Ordnern. Da wir uns aber für die generelle Verteilung interessieren hier die Grafik: Die beiden Grafiken zeigen uns, dass es viele Ketten gibt (speziell in Java) und dass viele Ordner in diesen Ketten sind. Außerdem können Ketten nicht nur ein oder zwei Ebenen tief sein sondern häufig auch 4 oder mehr Ebenen tief. Diese Erkenntnis wird in Abschnitt 7.6.1 eine Rolle spielen.

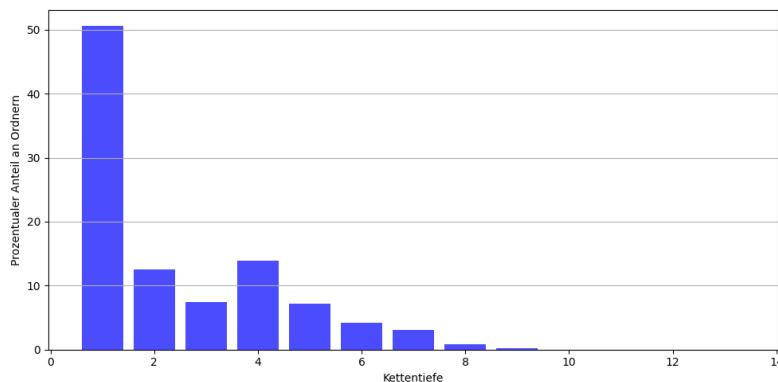


Abbildung 6.12: Die meisten Ketten sind nur 1 Ordner tief (Ordner + Ordner + Datei/Mehrere Dateien/Mehrere Ordner). Auffällig ist, dass es wieder Ketten von Länge 4 gibt.

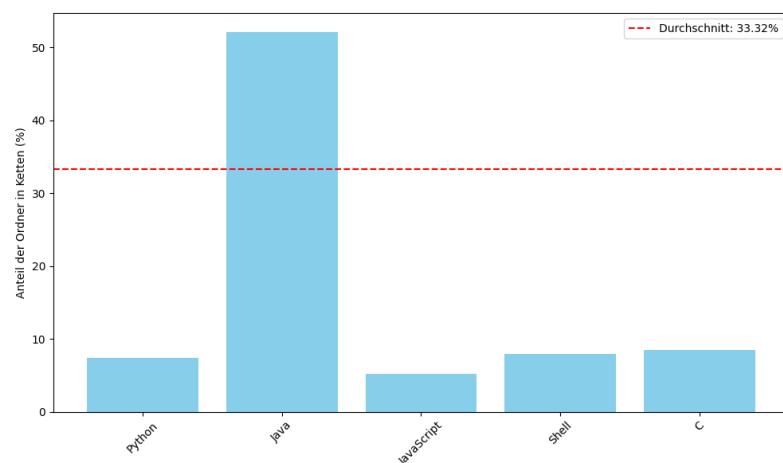


Abbildung 6.13: Die Anzahl der Ordner, die Teil einer Kette sind, ist besonder hoch bei Java. Der Durchschnittswert über alle Sprachen liegt bei ca. 33% und der von Java bei ca. 50%.

ERWEITERUNG DES SQUARIFY ALGORITHMUS

In diesem Abschnitt wird der Squarify Algorithmus [BHVWoo], wie er in Abschnitt 2.4.1 beschrieben wurde, auf verschiedene Weisen erweitert und angepasst, um das Layoutproblem, wie es in Abschnitt 3.2 beschrieben wurde, anzugehen.

7.1 *Kriterien für Treemap Layouts*

Um diese fünf Aspekte zu erreichen, definieren wir sechs Kriterien für das 2D layout, die diese Aspekte messbar machen. Diese Kriterien sollen helfen, die Qualität der Visualisierung zu bewerten und zu vergleichen. Die Kriterien sind:

- **Abstände:** Abstände zwischen den Knoten verbessern die Übersichtlichkeit und die visuelle Komplexität.
- **Knotensichtbarkeit:** Es sollte keine Knoten geben, die aufgrund von Abständen oder anderen Gründen nicht sichtbar sind. Dieses Ziel spielt speziell auf das in Abschnitt 2.4 beschriebene Problem ab.
- **Flächengröße:** Um die Korrelation mit dem Code zu gewährleisten, sollte die Fläche der Knoten proportional zum Metrikwert sein. (Hier ist noch unklar, ob das nur für Blätter gilt oder für alle Knoten)
- **Seitenverhältnis:** Um die visuelle Komplexität zu reduzieren und die Verständlichkeit zu erhöhen, sollte das Seitenverhältnis der Knoten möglichst nahe bei 1:1 liegen. Dies verbessert die Lesbarkeit der Knoten und macht es einfacher, die Informationen zu erfassen.
- **Platznutzung:** Es sollte so wenig Fläche wie möglich ohne Informationsgehalt bleiben. Als Gegenbeispiel kann man die Order-Knoten sehen, wie sie in der CodeCity Arbeit beschrieben wurden, bei denen die Fläche der Knoten nicht proportional zur Anzahl der Zeilen im Code ist, wodurch die Fläche an sich keinen Informationsgehalt mehr hat und außerdem viel leere Fläche entsteht.
- **Zeitaufwand:** Die Generierung des Layouts sollte in einem angemessenen Zeitrahmen erfolgen, um eine schnelle Visualisierung zu ermöglichen. Dies verbessert die Skalierbarkeit und generelle Nutzbarkeit der Visualisierung.
- **Stabilität:** Die Knoten sollten bei Änderungen die Position und Größe beibehalten, um eine stabile Visualisierung zu gewährleisten.

In dieser Arbeit sollen space filling approaches analysiert werden und speziell darauf untersucht werden, wie sie sich eignen für die definierten Anforderungen. Wie gut wird was erfüllt? Wann sollte man was anwenden? Kann eine gute Kombination aus verschiedenen Ansätzen gefunden werden? Bisher wurden im Grundlagenteil vor allem die splitting algorithmen vorgestellt, aber es gibt natürlich auch andere Ansätze, die verfolgt werden können, um Treemap Layouts zu generieren. Zum Beispiel gibt es bin packing oder optimierungs algorithmen. In dieser Arbeit sollen auch diese Ansätze betrachtet werden, um zu sehen, ob sie für die Visualisierung von Codequalitätsmetriken in einer space filling layout approach geeignet sind.

"Especially for the node weights, we assess if the algorithms use the weight values to scale the sizes of leaf nodes. Typically, the size of parent nodes is adjusted to the spatial consumption of the child nodes." [SLD20, S. 3] - Scheibel et al. sagen also, für sie ist es nicht so wichtig, dass die Größe der Elternknoten proportional zu den Metrikwerten der Knoten ist.

Dies soll getestet werden auf Basis von verschiedenen öffentlichen Repositories, die von kleinen bis großen Codebasen reichen. Als Metrik für die Fläche soll der Einfachehit halber die Anzahl der Zeilen verwendet werden.

auch schauen, wann der Algorithmus besonders gut und wann er schlecht ist - hängt natürlich von Inputdaten ab. z.B. ab welcher Tiefe ist er gut/schlecht. ab wie viel Knoten. Ab wie viel Knoten unterschied in der Größe

7.2 Zweifache Berechnung

Die Grundlegende Idee dieser Erweiterung ist es, dass sich die Fläche der Knoten mit Abstand durch das Layout und die Fläche der Knoten ohne Abstand approximieren lässt. Die Idee ist es also einen ersten Durchlauf zu machen, bei dem das Layout ohne Abstand berechnet wird. Dann werden die Größen der Knoten entsprechend dem Layout angepasst, sodass die Größe der Knoten nun auch den Abstand berücksichtigt. Anschließend wird ein zweiter Durchlauf mit diesen angepassten Größen durchgeführt, um das finale Layout zu berechnen.

Bevor wir uns die Details und Ergebnisse dieser Erweiterung anschauen, wollen wir vorweg nehmen, dass diese Erweiterung natürlich nicht optimal funktionieren kann und das auch klar ist, da sich die Änderung der Größe der Knoten natürlich auch das Layout im zweiten Durchlauf ändern wird, wodurch die Größen der Knoten wieder nicht korrekt sind. Was ja überhaupt erst das Grundlegende Problem ist (siehe Abschnitt 3). Allerdings ist es ein erster Schritt sich dem Problem zu nähern und zu schauen, ob es sich lohnt in diese Richtung weiter zu forschen.

Der Grundlegende Algorithmus bleibt also (fast) gleich, nur dass zwischen dem ersten und dem zweiten Durchlauf ein zusätzlicher Schritt Größenanpassung eingefügt wird. Die einzige Änderung die vorgenommen werden muss ist, dass Knoten nur mit dem definierten Abstand zwischen dem Elternknoten platziert werden können und generell die Fläche des Elternknotens um den Abstand verkleinert wird. Außerdem ist es nötig nach dem

zweiten Durchlauf die Knoten, deren Größenwert ja nun den Abstand beinhaltet, zu verkleinern, um auch den Abstand zwischen den Geschwistern herzustellen. Es ist zu erkennen, dass dadurch der Abstand sowohl zwischen Geschwistern als auch zu den Elternknoten den doppelten Wert des definierten Abstands hat, dieses Problem ignorieren wir hier, da man es trivialierweise lösen könnte, indem man immer nur die Hälfte des Abstands zwischen Geschwistern und Elternknoten abzieht, was wir hier der Einfachheit halber nicht tun. – ODER VIELLEICHT HIER IN DER THESIS DOCH? DANN KÖNNTE ICH MIR DIESEN ABSCHNITT SPAREN, AUCH WENN ES IN DER IMPLEMENTIERUNG AM ENDE ANDERS IST

Wir stellen verschiedene Ansätze vor, was sowohl die Größenanpassung als auch die Anpassung der Knoten nach dem zweiten Durchlauf angeht. Die Algorithmen funktionieren allerdings alle nach ähnlichem Prinzip: Es wird zunächst für jeden Knoten die Fläche die der Abstand in diesem Layout benötigen würde addiert, indem die Fläche aus der neuen Länge (alte Länge + 2 mal den Abstand) und der neuen Breite (alte Breite + 2 mal den Abstand) berechnet wird. Zusätzlich wird für Elternknoten die Flächenvergrößerung aller Kinderknoten addiert. An dieser Stelle ist allerdings nicht klar, wie sich die Flächenvergrößerung der Kinderknoten auf die Fläche der Elternknoten auswirkt, da diese Änderung selbst von der Anordnung der Kinderknoten abhängt. Wir testen verschiedene Ansätze, um die Flächenänderung der Elternknoten in Abhängigkeit zu der Flächenänderung der Kinderknoten zu approximieren.

Nach dem zweiten Durchlauf wird nun die Fläche der Knoten so reduziert, dass sowohl der Abstand zwischen Geschwistern als auch der Abstand zu den Elternknoten den gewünschten Wert hat. Dies ist straight forward und wird hier nicht weiter erläutert. Anzumerken ist aber das dieser Schritt speziell abhängt von der Art der Größenanpassung, die im ersten Schritt durchgeführt wurde.

7.2.1 *annäherungs wert*

Hier beschreiben, welche phi wert genutzt wurde und warum. Und das dann aus dem grundlagen teil raus nehmen

7.2.2 *Größenanpassung*

Dies ist die einfachste naive Version der Größenanpassung, der Algorithmus zeigt aber gut die zuvor beschriebenen Probleme auf. Die Fläche der Knoten wird um den Abstand in beiden Richtungen vergrößert. Zusätzlich wird die Fläche der Elternknoten um die Fläche der Kinderknoten vergrößert.

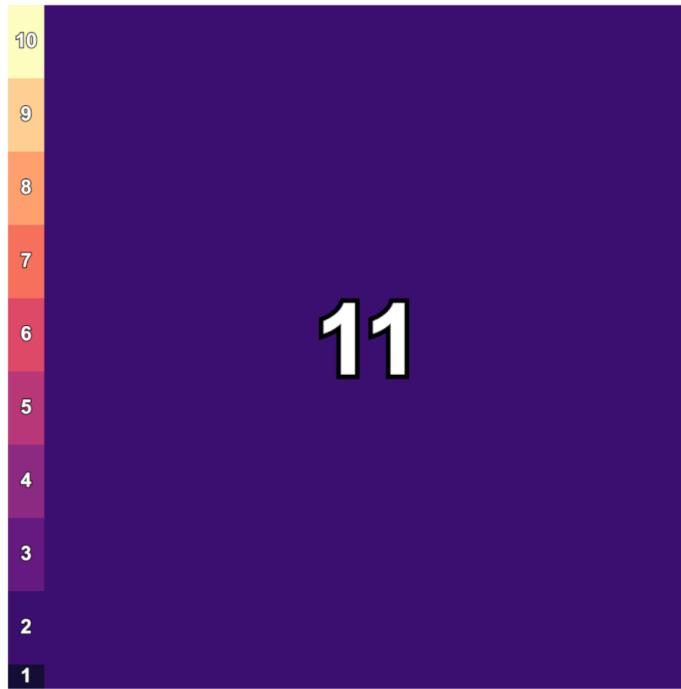


Abbildung 7.1: Treemap Layout generiert mit dem Squarify Algorithmus nach Abschnitt 2.4.1 mit einem Abstand von 0 und der einfachen Größenanpassung auf der händisch erstellen map (siehe Anhang).

Algorithm 1 Größenanpassung

```

1: function INCREASEVALUESIMPLE(node: SquarifyNode, margin: number)
2:   childrenValueIncrease ← 0
3:   if node.children then
4:     for child in node.children do
5:       childrenValueIncrease += increaseValuesSimple(child, margin)
6:     end for
7:   end if
8:   valueIncrease ← width * margin * 2 + length * margin * 2 + margin *
margin * 4 + childrenValueIncrease
9:   node.value += valueIncrease
10:  return valueIncrease
11: end function

```

Das Problem des Algorithmuses ist am besten an einem Beispiel zu verdeutlichen. Wir visualieren den ZWITEN AnHANG - auch wieder eine händisch erstellte Map, um das Problem zu verdeutlichen. In Abbildung 7.1 ist das Layout mit einem Abstand von 0 zu sehen.

In Abbildung 7.2 ist das Layout mit einem Abstand von 1 zu sehen. Es ist zu erkennen, dass die Knoten auf der linken Seite des Layouts sich außerhalb ihrer Elternknoten erstrecken. Warum passiert das? Knoten 10 ist im zweiten Layout-Schritt deutlich schmäler als im ersten Layout-Schritt. Dadurch wird die Fläche die durch den Abstand eingenommen wird größer als angenom-

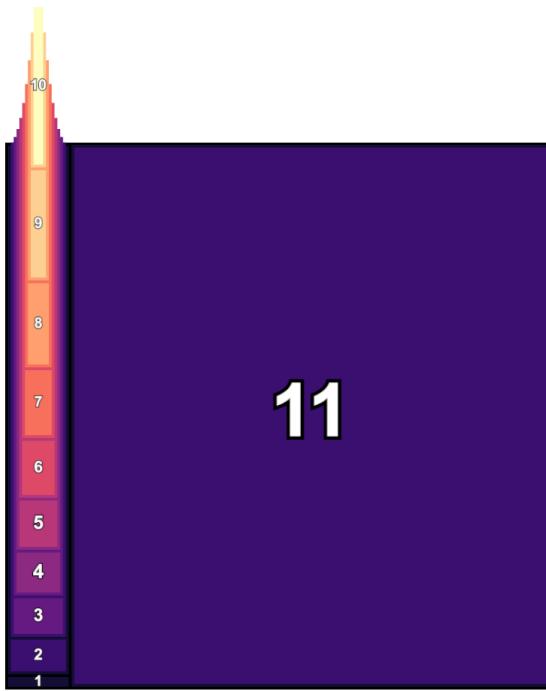


Abbildung 7.2: Treemap Layout generiert mit dem Squarify Algorithmus nach Abschnitt 2.4.1 mit einem Abstand von 1 und der einfachen Größenanpassung auf der händisch erstellen map (siehe Anhang).

men, weshalb die Fläche des Knotens nach abzug des Abstands im letzten Schritt kleiner ist, als gewünscht. Dementsprechend ist auch die Fläche der Knoten unten links größer als gewünscht, da das Layout der Knoten im zweiten Layout-Schritt quadratischer wird. Knoten 5, der Knoten, der am Quadratischsten ist, wird also am größten erscheinen. Obwohl beide den selben Wert haben ist Knoten 5 ca. 1,2 mal größer als Knoten 10) In diesem Beispiel erscheint der Unterschied kaum merklich, aber es gibt ihn trotzdem und in anderen Fällen kann dieser Unterschied merklich werden. Viel signifikanter ist aber der Effekt, dass Elternknoten ebenfalls immer schmäler werden, wodurch die Fläche, die der innerere Abstand einnimmt, ebenfalls größer wird und dass sogar immer mehr von Ebene zu Ebene, wenn man runter geht. Dadurch wird die Fläche für die Kindknoten immer kleiner, was dazu führt, dass Knoten teilweise über ihre Elternknoten hinauswachsen.

Dieser Effekt kann einfach behoben werden, wie wir in Abschnitt 7.3 sehen werden.

Bei der Berechnung zuvor wurde einfach die Marginfläche eines Elternknotens berechnet, auf Basis der alten Fläche, ohne die Flächenänderung durch die Kinderknoten zu berücksichtigen. Dadurch wird die resultierende Fläche der Elternknoten zu klein sein, da die Fläche der Kinderknoten nicht berücksichtigt wird. Wir stellen folgend eine zweite Version der Größenanpassung vor, die versucht die Fläche der Elternknoten durch die Fläche der Kinderknoten zu approximieren. Wir werden dann beide Versionen verglichen und schauen, welche besser funktioniert. Die Idee bei dieser Erweite-

rung ist es die Änderung der Kinder schon vor dem Hinzufügen der Abstandsfläche zu berücksichtigen. Dafür muss die relative Flächenänderung durch die Kindknoten berechnet werden. und damit dann die Seitenlängen anpassen und dann die margins hinzufügen, um die neue Fläche zu erhalten. Der Algorithmus wird im foldenden als Pseudocode dargestellt (siehe Algorithmus 2).

Algorithm 2 Relative Größenanpassung

```

1: function INCREASEVALUES(node: SquarifyNode, margin: number)
2:   width ← node.x1 - node.x0
3:   length ← node.y1 - node.y0
4:   if node.isLeaf then
5:     valueIncrease ← width * margin + length * margin + margin *
margin * 2
6:     node.value += valueIncrease
7:     return valueIncrease
8:   end if
9:   childrenValueIncrease ← 0
10:  if node.children then
11:    for child in node.children do
12:      childrenValueIncrease += increaseValuesRelative2(child, mar-
gin)
13:    end for
14:  end if
15:  ratioChildrenValueIncrease ← (node.value + childrenValueIncrease) /
node.value
16:  valueIncrease ← Math.sqrt(ratioChildrenValueIncrease) * width *
margin * 2 + Math.sqrt(ratioChildrenValueIncrease) * length * margin
* 2 + margin * margin * 4 + childrenValueIncrease
17:  node.value += valueIncrease
18:  return valueIncrease
19: end function

```

Problem: Der Algorithmus in dieser Form zeigt einige Probleme auf. Die Flächenvergrößerung der Kindknoten sagt nichts darüber aus, in welche Richtung sich die Fläche ändert. Es wird davon ausgegangen, dass sich die Fläche gleichmäßig in beide Richtungen ändert (siehe Zeile 13 und 14 in Algorithmus ZEILEN ANPASSEN 2). es kommt also zu ähnlichen Problem wie davor. es können knoten sowohl zu viel platz als auch zu wenig Platz bekommen, jenachdem ob Sie im zweiten schritt quadratischer oder schmaler werden. Siehe Abbildung 7.3 für ein Beispiel.

Evaluation: mit diesen Werten:

7.3 Scaling der Knoten

Wenn die Fläche innerhalb von Elternknoten immer kleiner wird, kann es passieren, dass Knoten über ihre Elternknoten hinauswachsen, wie in Abbil-

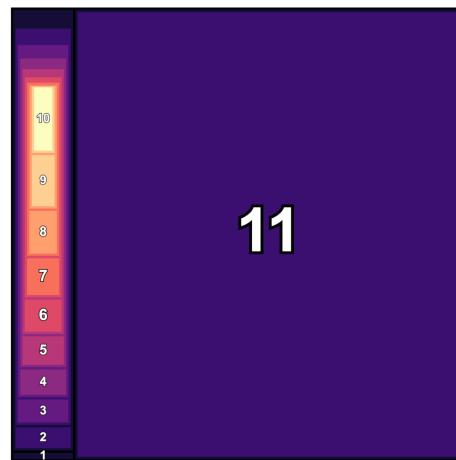


Abbildung 7.3: Treemap Layout generiert mit dem Squarify Algorithmus nach Abschnitt 2.4.1 mit einem Abstand von 1 und der relativen Größenanpassung auf der händisch erstellen map (siehe Anhang).

dung 7.2 zu sehen ist. Es kann genauso passieren, dass die Fläche innerhalb der Knoten größer wird, wie in Abbildung 7.3 auf der linken Seite zu erkennen ist. Dieser Effekt kann trivialer weise behoben werden, indem der zweite Layout-Schritt angepasst wird, sodass die Knoten immer auf die Fläche des Elternknotens skaliert werden. Vor jeden Squarify-Schritt wird dafür die wirklich zur Verfügung stehende Fläche des Elternknotens berechnet und die Kindknoten entsprechend dieser Änderung skaliert, sodass sie genau in die Fläche des Elternknotens passen.

Der Nachteil dieser Methode ist in Abbildung 7.4 zu erkennen. Die Knoten werden dadurch natürlich nicht mehr proportional zu ihren Werten sein. Knoten 10 hat zum Beispiel ein Verhältnis von ca. 0.5 zu seinem Wert, während Knoten 6 ein Verhältnis von ca. 1.4 zu seinem Wert hat.

Je genauer die Größenanpassung der Knoten ist, desto geringer fällt natürlich dieser Effekt aus. Siehe im Vergleich dazu Abbildung 7.5, da die relative Größenanpassung deutlich genauer ist, ist der Effekt hier auch deutlich geringer. Knoten 10 hat Größe 40 und Knoten 6 hat Größe 41, was fast ähnlich groß ist.

Persönlich würde ich sagen, dass dieser Effekt für das herunter skalieren der Knoten gut ist, da sonst eine grundlegende Eigenschaft der Darstellung verletzt wird, aber für das hoch skalieren der Knoten könnte man auch sagen, dass es wichtiger ist die Fläche der Knoten proportional zu ihren Werten zu halten, als die *verschwendete* Fläche aufzufüllen. Diese jetzt hier sehr subjektive Einschätzung wird aber nochmal genauer beläuchtet in der Evaluation und vergleich.

7.4 Reihenfolge der Knoten im ersten Layoutschritt

Die Reihenfolge in der die Knoten in das Layout eingefügt werden, hat einen großen Einfluss auf das Layout. [JS91] haben in ihrem Paper herausgefunden,

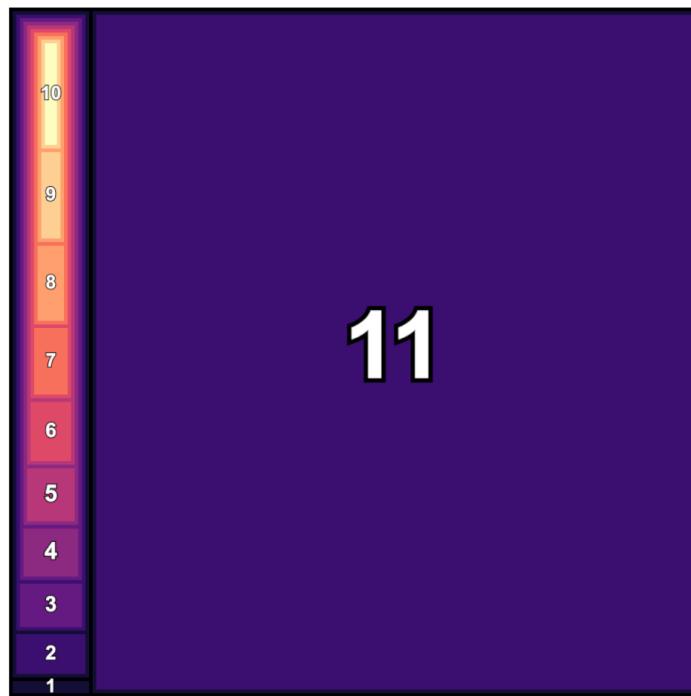


Abbildung 7.4: Treemap Layout generiert mit dem Squarify Algorithmus nach Abschnitt 2.4.1 mit einem Abstand von 1 und der einfachen Größenanpassung auf der händisch erstellen map (siehe Anhang) und der Skalierung der Knoten.

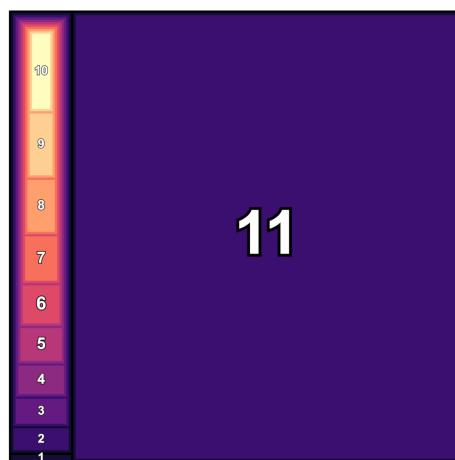


Abbildung 7.5: Treemap Layout generiert mit dem Squarify Algorithmus nach Abschnitt 2.4.1 mit einem Abstand von 1 und der relativen Größenanpassung auf der händisch erstellen map (siehe Anhang) und der Skalierung der Knoten.

den, dass die ergebnisse am besten sind, wenn die Knoten in der Reihenfolge der Größe eingefügt werden. Also der Größte zuerst. Außerdem haben wir in Abschnitt 6.2 gezeigt, dass 95% der Ordner maximal 18 Kindknoten haben. Das sortieren aller Knoten in einem Ordner nach Größe ist also gar nicht mal so Zeitaufwendig, wie man denken könnte, wenn man tausende von Knoten hat, weil immer nur ein kleines Set von Knoten sortiert werden muss. Kleines Beispiel: Wenn wir von einem typischen Sortieralgorithmus mit einer Laufzeit von $\mathcal{O}(n \log n)$ (z.B. Quicksort), einer Anzahl von 4221 Knoten (Durchschnittswert der Anzahl der Knoten siehe Abschnitt 6.2) und 18 Knoten pro Ordner. Wenn wir alle Knoten auf einmal sortieren wollen:

$$T_1 = n \cdot \log_2(n) = 4221 \cdot \log_2(4221)$$

$$\log_2(4221) \approx 12.04$$

$$T_1 \approx 4221 \cdot 12.04 \approx 50821$$

Wenn wir die Knoten in 18er Bundeln sortieren wollen:

$$T_2 = m \cdot (k \cdot \log_2(k)) = 235 \cdot (18 \cdot \log_2(18))$$

$$\log_2(18) \approx 4.17$$

$$T_2 \approx 235 \cdot (18 \cdot 4.17) = 235 \cdot 75.06 \approx 17639$$

Der Algorithmus mit Sortierung braucht im Schnitt ca. 5.5ms und mit Sortierung ca. 6.5ms. also ca. 1ms mehr (Median 0.4ms).

7.4.1 Reihenfolge der Knoten nach erstem Layoutschritt

Durch die Updates der Größen der Knoten nach dem ersten Layoutschritt, kann es passieren, dass die Größenreihenfolge der Knoten sich ändert. Das als zum Beispiel Knoten A im ersten Layoutschritt größer ist als Knoten B, aber nach dem Update der Größen von Knoten A kleiner ist als Knoten B. Das kann dazu führen, dass im folgenden Layoutschritt, die Knoten in einer anderen Reihenfolge platziert werden als im ersten Layoutschritt. Das ist erstmal gut für die Seitenverhältnisse der Knoten (wenn wir davon ausgehen, dass die Reihenfolge nach Größe sortiert optimiert ist), allerdings kann es dazu führen, dass Knoten an ganz anderen Positionen platziert werden, als im

ersten layoutschritt. Das kann dazu führen, dass sich die Form ändert und das kann dazu führen, dass die Kinder anders platziert werden etc. wodurch sich die benötigten Abstände ändern, wodurch die vorausberechnete Fläche der Knoten nicht mehr passt und Knoten entweder verschwinden oder die Wert Fläche relation schlechter wird. Das kann natürlich genauso passieren, wenn die Knoten in der selben Reihenfolge platziert werden, wie im ersten Layoutschritt. Es ist nicht klar erstichtlich, was hier am Ende des Tagen am besten ist. Deswegen implementieren wir 3 verschiedene Ansätze:

- **Reihenfolge der Knoten nach Größe:** Die Knoten werden in der Reihenfolge ihrer Größe platziert. Wir vermuten, dass dadurch die besten Seitenverhältnisse entstehen, aber die schlechtesten WErt-Fläche-Relationen.
- **Reihenfolge der Knoten beibehalten:** Die Knoten werden in der Reihenfolge platziert, in der sie im ersten Layoutschritt platziert wurden. Wir vermuten, dass dadurch im Großen und Ganzen die plazierung der Knoten gleich bleibt, außer es gibt wirklich große Flächen updates. Wir vermuten, dass dies ein guter mitelweg zwischen dem ersten und dem jetzt folgenden ansatz ist.
- **Platzierung der Knoten beibehalten:** Nach dem ersten Layoutschritt wird sich genau die Aufteilung der Knoten in Reihen und die Platzierung der Knoten in den Reihen gemerkt. Wir vermuten, dass dadurch die Form der Knoten im großen und ganzen ähnlich zum ersten Schritt bleibt, wodurch die Wert-Fläche-Relationen besser bleibt, aber die Seitenverhältnisse schlechter werden könnten, die die optimale Reihenfolge der Knoten (aufgrund der neuen werte) nicht mehr berücksichtigt wird.
-

Es ist wie gesagt nicht ganz klar, was genau die auswirkungen sind. deswegen nur vermutungen, die wir jetzt testen werden.

7.5 Mehrfache Berechnung

Die Grundlegende Idee dieser Erweiterung ist es, das Layout mehrfach zu berechnen und dabei die Fläche der Knoten immer weiter anzupassen. Warum glauben wir, dass das gut ist. nach dem ersten update schritt ist die vorher-sage der neuen fläche ja oft nicht so gut, und es kann sich noch einiges in der reihenfolge der Knoten ändern oder die Form der Knoten. dadurch ist nach dem zweiten Layout-Schritt die platzierung ja anders, dadurch stimmt die vorhergesagte Fläche der Knoten nicht mehr. dann könnte man nochmal her gehene und die fläche neu berechnen und nochmal den layoutschritt machen.

Grund dafür: Grund dagegen:

7.5.1 Iterative Abstandsvergrößerung

In bild updateValues.png ist zu erkennen, dass kp. margin 10 noP 3

Die Grundlegende Idee dieser Erweiterung ist es, das wenn das Layout mehrfach berechnet wird, es besser ist, den Abstand zwischen den Knoten iterativ von Schritt zu Schritt zu vergrößern, anstatt ihn von Anfang an auf den finalen Wert fest zu setzen. Warum gehen wir von dieser Annahme aus? Wenn man im ersten Schritt den Abstand auf den finalen Wert setzt, dann werden die Vorhersagen für die neue Fläche der Knoten im zweiten Schritt nicht exakt sein, das ist ja klar. Umso kleiner die Abstände sind, umso kleiner sind auch die Abweichungen dieser Vorhersagen. Dadurch sollten die Wert-Fläche-Relationen im zweiten Schritt besser sein. Die Annahme ist jetzt, dass wenn man jedes mal den Abstand um einen kleinen Wert vergrößert, dann sind in jedem Schritt die Vorhersagen für die Fläche der Knoten genauer, wodurch die Wert-Fläche-Relationen besser werden. Als Gegenargument könnte man bringen, dass dadurch, dass jedes mal die Abstände größer werden auch jedes mal das Optimale Layout (man kann das optimale layout nicht berechnen, aber wir gehen mal von einem theoretschen optimalen Layout aus) auch in jedem layout schritt anders sein wird. Dadurch wird erst im letzten Layoutschritt das "wahre optimale Layout angestrebt, wodurch die vorherigen schritte unnötig waren.

Wir überprüfen jetzt die annahme und schauen, welche begründung richtiger ist/welcher effekt mehr gewicht hat. hier ist es auch interessant den effekt in kombination mit der Reihenfolge der Knoten zu testen, weil

7.6 Änderungen im Layout

Folgend werden Änderungen beschrieben, die sich sichtbar auf die Visualisierung auswirken und nicht nur auf Platzierung, Größe, Form,... etc der Knoten. Diese Änderungen zielen trotzdem vor allem darauf ab, das Treemap-Layout-Problem zu lösen bzw. abzuschwächen, machen dies aber vor allem durch visuelle Anpassungen und nicht durch komplexe algorithmische Anpassungen.

7.6.1 Optimierung von Ordnerketten

Bei der Datenanalyse ist aufgefallen, dass es oft vorkommt, dass Ordner nur ein Kind haben bzw. dass es *Ketten* von Ordnern gibt, die nur ein Ordner als Kind haben. Häufig sind diese Ketten auch nicht nur einen Ordner tief, was gar nicht so viel Abstand bzw. platz verbrauchen würde, doch es gibt auch einige ketten die 4 ordner tief sind, was dann schon wieder (mehr) als viel mal so viel platz für abstände auf braucht durch die Verschachtelung. obwohl im Grunde diese Schachtelung keine wirklichen Strukturinformationen enthält. Viele IDEs wie zum Beispiel auch IntelliJ IDEA oder VSCode verbergen diese Ordnerstruktur zum beispiel auch automatisch, in dem solche ordner ketten nur als ein Ordner mit entsprechendem zusammen gefügten namen (in dem fall *main/kotlin/de/maibornwolff/codecharta/analysers/-*

analyserinterface) angezeigt wird. Diese Optimierung kann auch für unsere Visualisierung übernommen werden, indem solche Ordnerketten zu einem Ordner zusammengefasst werden. Es ergeben sich folgende Auswirkungen auf die Ziele der Visualisierung:

- **Informationsgehalt und effiziente Nutzung des Platzes:** Wird verbessert, da weniger Platz für Abstände zwischen den Ordner benötigt wird und weniger Knoten im Layout sind.
- **Niedrige visuelle Komplexität und hohe Verständlichkeit:** Wird verbessert, da weniger Knoten im Layout sind und tiefe Verschachtelungen vermieden werden.
- **Skalierbarkeit:** Wird verbessert, da weniger Knoten im Layout sind und die Berechnung des Layouts beschleunigt wird.
- **Korrelation mit dem Code:** wird verschlechtert, da die Ordnerstruktur nicht mehr 1:1 mit der Code-Struktur übereinstimmt. die *wahre* Tiefe der Ordnerstruktur nicht mehr ersichtlich ist. Allerdings ist die Tiefe der Ordnerstruktur selbst in eigentlich gar nicht so relevant, die Tiefe wird eher als Mittel benutzt um die Ordner Struktur darzustellen. Ob eine Datei jetzt 5 oder 10 tief liegt, macht für die Wahrnehmung der Struktur keinen großen Unterschied.
- **Stabilität gegenüber Änderungen:** Wird verbessert, da durch weniger Abstände die Differenz zwischen Knoten Fläche und Knoten Wert geringer werden, wodurch weniger Umstrukturierung bei der Platzierung der Knoten nötig ist. Außerdem wirken sich trivialerweise Änderungen an Order-Ketten nicht mehr auf die gesamte Struktur aus und verändern somit auch die Visualisierung nicht, was positiv für die Stabilität ist (aber wie gesagt negativ für die Korrelation mit dem Code).

Wir würden sagen, dass die Vorteile dieser Änderung die Nachteile stark überwiegen.

Im Schnitt werden ca. 175 Ordner pro Codebasis gefaltet (12 Median). bei ca. 958 Knoten (Median 242) sind das ca. 18% (bzw. 5% Median) der Knoten die durch die Faltung verschwinden. Also schon einige.

Folgend kommen die DAten:

die Daten Zeigen in allen Aspekten eine Verbesserung durch das Falten von Ordnerketten. Wir sprechen uns also dafür aus, dass Ordnerketten immer gefaltet werden sollten, von den messbaren Kriterien her sehen wir keine Gründe, warum das nicht gemacht werden sollte. das einzige Ziel was dagegen spricht ist die Korrelation mit dem Code, die dadurch verschlechtert wird. Alles in allem sprechen wir uns also dafür aus und werden dies fix für unseren Algorithmus implementieren.

7.6.2 Beschriftung der Knoten

Die Beschriftung der Knoten ist ein wichtiger Aspekt der Visualisierung, da sie es dem Nutzer ermöglicht, die Knoten zu identifizieren und zu verstehen,

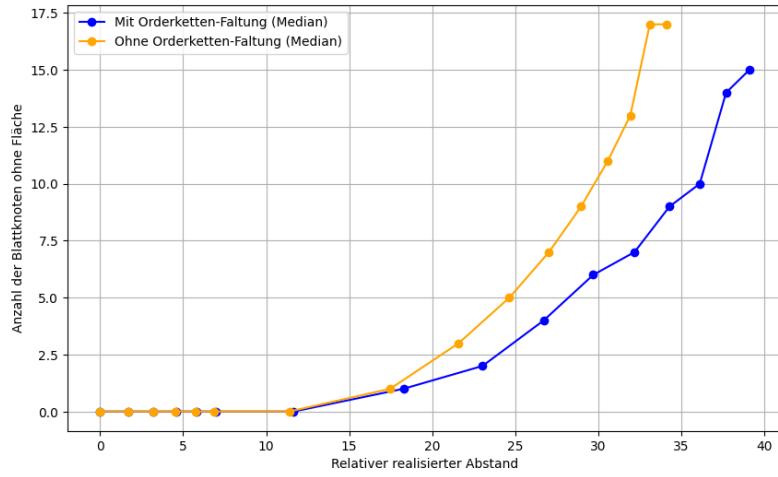


Abbildung 7.6: Die Anzahl der Knoten ohne Fläche geplottet gegenüber des Abstandes. Es ist zu erkennen, dass das Falten von Ordnerketten die das verschwinden von Knoten reduziert.

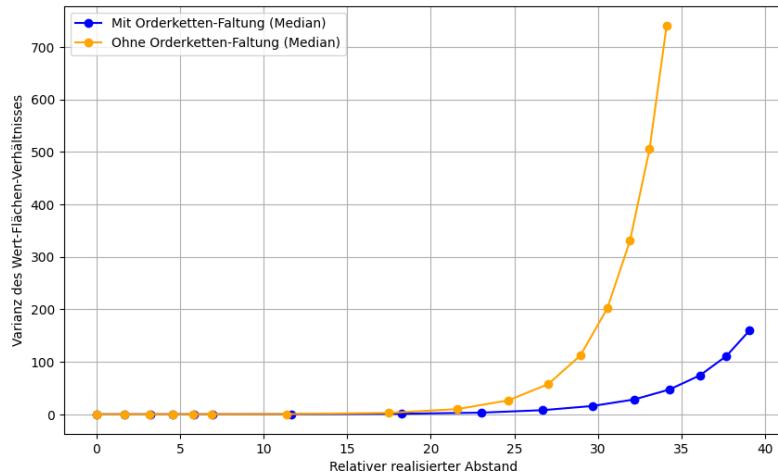


Abbildung 7.7: Das Wert-Fläche-Verhältnis der Knoten geplottet gegenüber des Abstandes. Es ist zu erkennen, dass das Falten von Ordnerketten das Wert-Fläche-Verhältnis der Knoten verbessert.

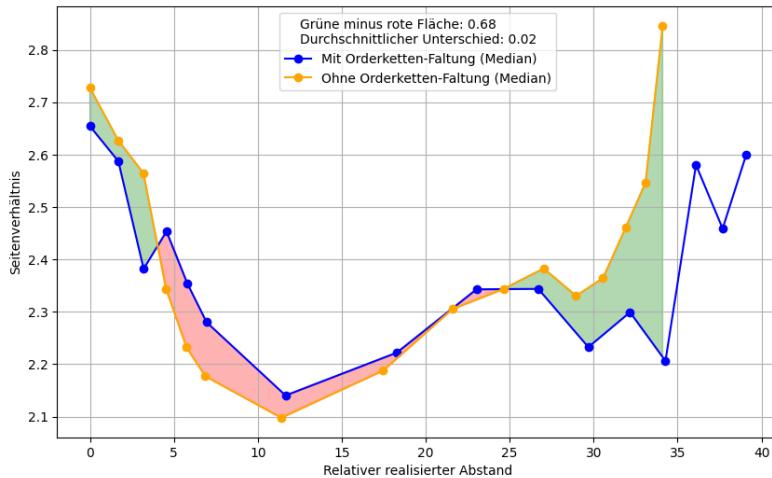


Abbildung 7.8: Das Seitenverhältnis der Knoten geplottet gegenüber des Abstandes. Es ist nicht klar zu erkennen, ob das Falten von Ordnerketten das Seitenverhältnis der Knoten verbessert oder verschlechtert, deshalb sind die Bereiche des Positiven effekts grün markiert und die Bereiche des negativen Effekts rot markiert. Es zeigt sich ein leicht positiver Effekt besonders bei kleinen und großen Abständen.

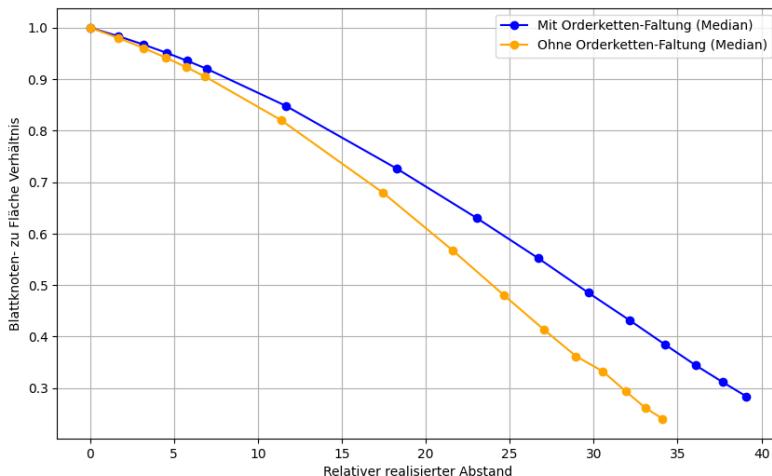


Abbildung 7.9: Die Fläche der Blattknoten in Verhältnis zu der gesamt Fläche geplottet gegenüber des Abstandes. Es ist zu erkennen, dass das Falten von Ordnerketten die Fläche der Blattknoten erhöht. Das heißt, es wird weniger Platz benötigt für Abstände und Ordner, also gleiche Informationen auf weniger Fläche, bedeutet besserer Informationsgehalt.

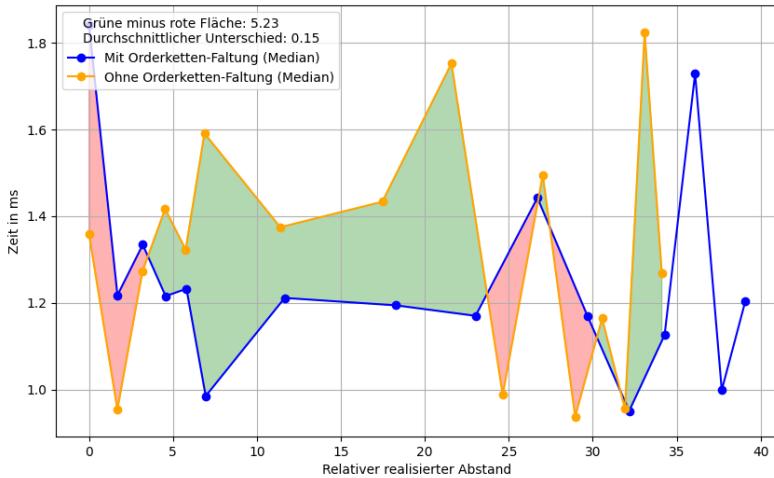


Abbildung 7.10: Die Rechenzeit geplottet gegenüber des Abstandes. Es ist nicht klar zu erkennen, ob das Falten von Ordnerketten die Rechenzeit verbessert oder verschlechtert, deshalb sind die Bereiche des Positiven Effekts grün markiert und die Bereiche des negativen Effekts rot markiert. Die Schwankungen sind zwischen ungefähr 1 und 2 millisekunden, was nicht wirklich ausschlaggebend ist. Trotzdem ist ein leichter performance gewinn zu erkennen beim falten, was wahrscheinlich einfach daran liegt, dass es weniger Knoten gibt.

welche Informationen sie repräsentieren. Es gibt verschiedene Ansätze, um die Beschriftung der Knoten zu gestalten. Jeder Ansatz hat seine Vor- und Nachteile. Ein häufig verwendeteter Ansatz ist wie zum Beispiel auch beim in Abschmijt VerwandteArbeiten 4 vorgestellten Cascaded Treemap Ansatz [LFo8]. Ordner erhalten eine Zusätzliche horizontale Fläche, entweder unten oder oben, die den Namen des Ordners enthält. Dies nimmt allerdings viel Platz ein und erzeugt die selben schwierigkeiten für den Algorithmus, wie die abstände zwischen den Knoten. Außerdem kann es passieren, dass die Beschriftung zu lang für die breite der Knotens ist, wodurch die Beschriftung entweder abgeschnitten wird oder über den Knoten hinauswächst. Hao Lü and James Fogarty schlagen vor dynamisch zu entscheiden, ob die Beschriftung der Knoten angezeigt werden soll oder nicht [LFo8]. Wenn das Hinzufügen der Beschriftung den Knoten dazu führen würde, dass gar kein Platz mehr für den Knoten selbst übrig bleibt, wird die Beschriftung nicht angezeigt. Der Vorteil dieses Ansatzes ist, dass sicher gegangen wird, dass durch die Beschriftung kein Knoten verschwindet. Der Nachteil ist, dass selbst wenn die Änderung der Knotenfläche zum Knotenwert durch das hinzufügen der Beschriftung riesig wird (wenn die Fläche also super klein wird), die Beschriftung trotzdem hinzugefügt wird.

Wir schlagen einen statischen ansatz vor, bei dem nur die Beschriftungen der Top N order angezeigt werden. Die Wahl von N ist nun die Stellschraube für die nicht für jede Codebasis gleich ist. Es ist Stellschraube um zwischen Informationsgehalt und visuelle Komplexität zu balancieren.

- Je höher N ist, desto mehr Informationen (also Ordnernamen) werden angezeigt, was den Informationsgehalt erhöht.
- Je niedriger N ist, desto weniger Informationen werden angezeigt, was die visuelle Komplexität reduziert, viele Beschriftungen können schnell ablenken und die Visualisierung unübersichtlich machen.
- Außerdem wird die Korrelation mit dem Code verbessert, da ein übertragen von Vsiaulisierung zur codebasis einfacher wird mit höherem N.
- Skalierbarkeit hängt nicht ab von N, da N ja nicht mitwächst, sondern statisch ist, ob noch mehr order und noch mehr tief hinzu kommt, macht keinen wirklichen unterschied.
- STabi gegen änderungen wenn überhaupt sinkt es, weil änderung vom namen zb. auf einmal sichtbar wird (das ist natürlich gewollt, trotzdem in unserer messung hier erstmal negativ- wenn auch vernachlässigbar)
 - eventuell werden dadurch in zwei versionen die selben Knoten, die jetzt anders heißen, nicht mehr so einfach als gleich identifiziert, obwohl diese die selbe form haben und ohne die beschriftung vielleicht anhand der form als gleich identifiziert werden würden - zugegeben sehr theoretisch und wahrscheinlich nicht relevant aber trotzdem erwähnenswert.

Mit diesem Ansatz hat man natürlich immernoch die gleichen Probleme wie eingangs beschrieben, allerdings sind diese abhängig von dem wert N und außerdem werden diese minimiert, da wirklich nur die Top Knoten beschriftet werden, wodurch die Fläche der unteren Knoten trotzdem besser vorhergesagt werden kann und sich dieser fehler nicht nach oben propagiert.

7.6.3 Abstände zwischen Geschwister Knoten

Abstände zwischen Geschwister Knoten sind eine wichtige Entscheidung. Im Ursprünglichen Squarify Algorithmus Paper [BHVWoo] wird der Nested Ansatz mit Abständen vorgestellt, jedoch ohne Abstände zwischen Geschwistern. Viele Tools, verwenden jedoch Abstände auch zwischen Geschwisterknoten [WLo7; Gmbc; Gmba]. Was sind die Vor- und Nachteile von Abständen zwischen Geschwistern?

- **Informationsgehalt und effiziente Nutzung des Platzes:** Wird verschlechtert, da jeder Knoten mit geschwistern mehr Platz benötigt.
- **Niedrige visuelle Komplexität und hohe Verständlichkeit:** Wird verbessert, da die Knoten besser voneinander getrennt werden und somit die Struktur besser erkennbar ist. Allerdings muss das nicht der Fall sein, man könnte zum Beispiel die Kanten der Knoten mit einer anderen Farbe einfärben, um die Knoten besser voneinander zu trennen.

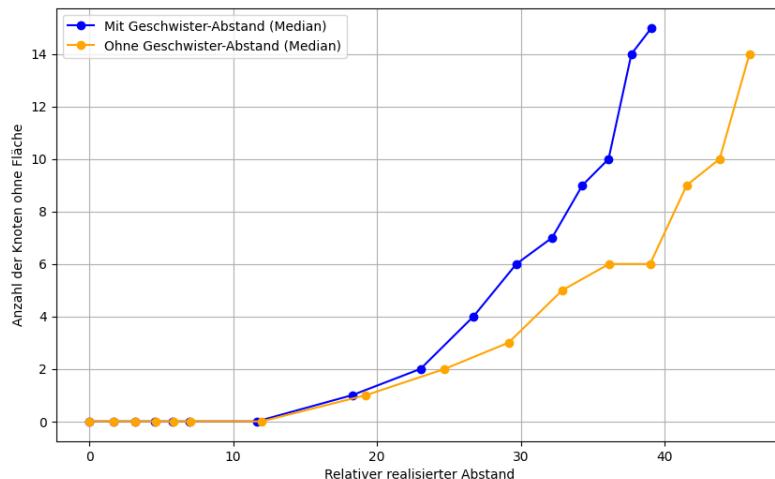


Abbildung 7.11: Die Anzahl der Knoten ohne Fläche geplottet gegenüber des Abstandes.

- **Skalierbarkeit:** Eigentlich wenig auswirkung auf skalierbarkeit, weder die laufzeit ändert sich signifikant (in unserem algorithmus werden die abstände erzeugt, indem am ende alle knoten um den abstand verkleinert werden - also im grunde kommt laufzeit von N hinzu, wo N die Anzahl der Knoten ist, das ist aber nicht ausschlaggebend). Also wenn wird die Skalierbarkeit minimal schlechter.
- **Korrelation mit dem Code:** Keine auswirkung
- **Stabilität gegenüber Änderungen:** Auch nicht wirklich

Die Anpassung, die Knoten mit Kanten zu versehen, natürlich stark abhängig von der gewählten Farbgebung (auch Schattierung, Struktur oder Transparenz). Wie in der Problemstellung definiert (siehe Abschnitt 3) sollte das Layout so gestaltet sein, dass es möglich ist, weitere Metriken durch Farbgebung zu visualisieren. Die Kanten widersprechen dieser Einschränkung nur in Aussahme fallen. Die meisten Visualisierungen in der Praxis verwenden einfache Einfärbungen [WLo7; Gmbc; Gmba], was bedeutet, dass man einfach Schwarz oder eine Komplementärfarbe als Kantenfarbe nutzen kann, ohne dass es zu Konflikten mit der Farbgebung kommt. Die einzige Ausnahme, die mir bekannt ist, ist wäre die Nutzung von Linien, Schachbrettmuster oder ähnlichen Strukturen, durch die eine Kante nicht mehr als solche erkennbar wäre. Dies könnte zum Beispiel bei dem Ansatz aus dem paper *E-Quality* von Ural Erdemir et al. [ETB11] der Fall sein, da hier zum Beispiel ein Gittermuster verwendet wird, um einen Mangel an Kohäsion zu visualisieren, eine Kante, selbst in anderer Farbe, würde hier eher stören oder in dem Muster untergehen.

7.7 Fazit

Immernoch straight forward, es gibt aber noch Probleme

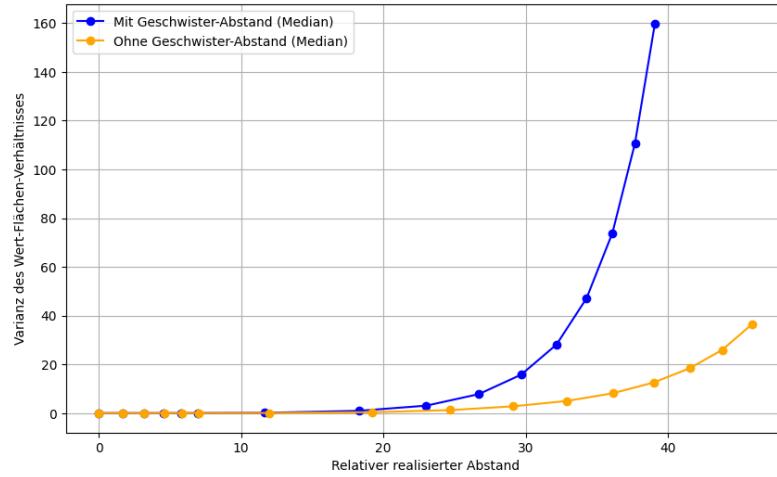


Abbildung 7.12: Das Wert-Fläche-Verhältnis der Knoten geplottet gegenüber des Abstandes. Es ist zu erkennen, dass die Abstände zwischen Geschwistern das Wert-Fläche-Verhältnis der Knoten verbessert.

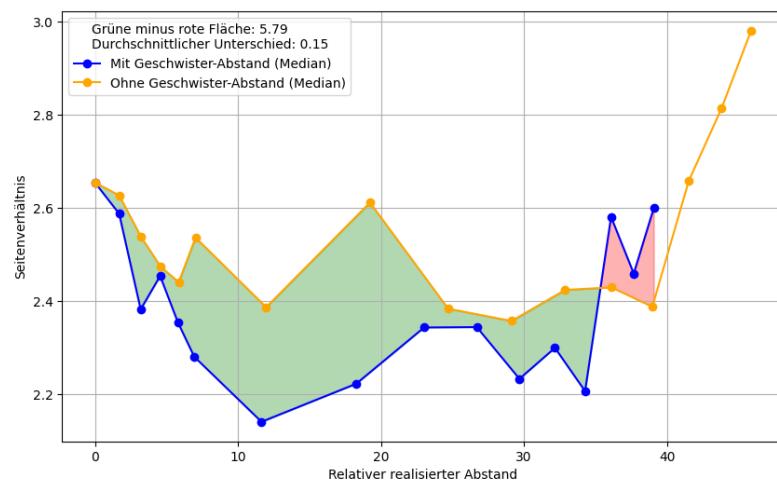


Abbildung 7.13: Das Seitenverhältnis der Knoten geplottet gegenüber des Abstandes. Es ist zu erkennen, dass die Abstände zwischen Geschwistern das Seitenverhältnis der Knoten verbessert.

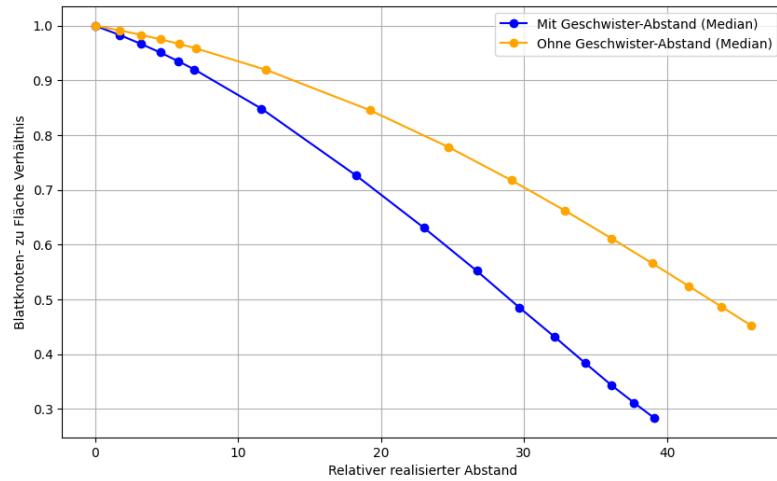


Abbildung 7.14: Die Fläche der Blattknoten in Verhältnis zu der gesamt Fläche geplottet gegenüber des Abstandes. Es ist zu erkennen, dass die Abstände zwischen Geschwistern die Fläche der Blattknoten erhöht. Das heißt, es wird weniger Platz benötigt für Abstände und Ordner, also gleiche Informationen auf weniger Fläche, beduetet besserer Informationsgehalt.

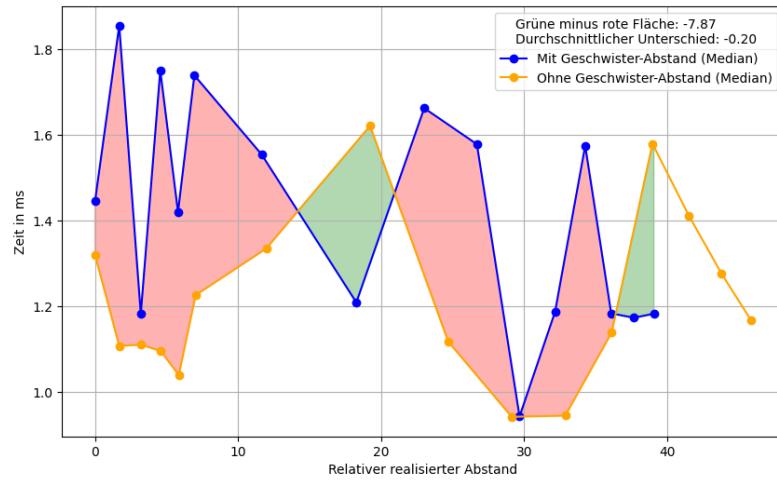


Abbildung 7.15: Die Rechenzeit geplottet gegenüber des Abstandes. Es ist zu erkennen, dass die Abstände zwischen Geschwistern die Rechenzeit nicht signifikant beeinflussen.

Warum besonders den squarify algorithmus betrachtet und nicht zum beispiel pivot oder circle?? -> weil diese Andere Ziele haben -> noch mehr begründen anhand der definierten Ziele

8

EVALUATION

Evaluation anhand eines echten beispiels mit verschiedenen Layouts. In diesem Abschnitt werden 1. die verschiedenen Treemap-Algorithmen verglichen und anhand von konkreten Kriterien evaluiert, um den besten Algorithmus für die Visualisierung von Codequalitätsmetriken zu finden. 2. Werden alle vorgestellten Layouts anhand eines konkreten Beispiels aus der Praxis evaluiert. Dies wird gemacht, indem eine echte, bereits durchgeführte, Software-Analyse nochmal mit den verschiedenen Layouts visualisiert wird. Dann wird verschiedenen unerfahrenen Personen diese verschiedenen Visualisierungen gezeigt und gefragt, wie sie die Software-Qualität einschätzen würden. Wenn das möglichst nahe an die Experten Einschätzung kommt, so wie es in der Analyse durchgeführt wurde, dann ist das Layout gut geeignet.

FAZIT

In dieser arbeit konnte natürlich ein eine endliche Anzahl an Ansätzen und Ideen untersucht werden. In Zukunft ist bestimmt noch viel kreativität und Forschungspotential in diesem Bereich. Außerdem vielleicht sogar komplett neue Ansätze, die nicht auf der Stadt-Metapher basieren sondern vielleicht auf planeten oder anderen Metaphern, auch wenn zu vermuten ist, dass dabei die übersichtlichkeit und einfachheit leiden könnte. Die in diser arbeit untersuchten Ansätze sind alle sehr einfach mit basic formen und daher übersichtlich.

Es ist natürlich klar, dass der treempa ansatz optmiert wurde und die anderen Ansätze nicht, weshalb der treemap ansatz besere ergebnisse liefert. Es bleibt zu erforschen, ob sich die anderen Ansätze auch so optimieren lassen, dass sie bessere Ergebnisse liefern.

Außerdem ist bei der literatur recherche noch interessant auch komplett über den tellerrand von software hinaus zu schauen und andere hierarchical vis zu betrachten, die nicht direkt mit software zu tun haben, um eventuell neue gut viz zu finden. Oder sogar ganz anders mit Kreativen ideen findungs ansätzen komplett neue Ansätze zu finden. Oder sich neue metaphern zu überlegen und nicht nur stadt, inseln, weltall,....

Außerdem ist es interannt in der umfrage die tools den nutzern direkt zur verfügung zu stellen. In dieser arbeit wurden tools nur mit statischen bildern vorgestllt, um sich konkret nur auf die Visualisierung zu konzentrieren und faktoren wie nutzerfreundlichkeit, performance und interaktion nicht mit zu betrachten.

Äußerdem kann es passieren, dass die Beschriftung zu lang für die breite der Knotens ist, wodurch die Beschriftung entweder abgeschnitten wird oder über den Knoten hinauswächst."das angehen, mit zum beispiel rechteck breiter machen oder so, anders paltzieren, PIPAPO

Außerdem eventuell dynamische Beschriftung implementieren

Auch interessant wenn die struktur oder die metriken nicht auf file, sondern zum beispiel auf klassen basis sind. schauen, ob das einen unterschied macht.

Bewusst, dass gewisse wissenschaftliche standards nicht zu 100 Przent eingehalten wurden Zb. hat nur eine person am code gearbeteitet, es wurde speziell der algorithmus so lange implementiert, bis die ergebnisse korrekt aussahen, das ist keine 100 prozentige garantie, dass zb die evaluation wirklich korrekt und fehlerfrei ist. interessant wäre nochmal eine arbeit, die den code überprüft. aber bei software engineering immer so, dass es bugs und fehler gibt, nie auszuschließen. aber in dieser arbeit größte vorsicht und gewissenhaftigkeit - auch wenn das nicht wissenschaftlich ist. Deswegen stellen wir den kopmletten quellcode zur verfügung, damit andere die Ergeb-

nisse überprüfen können. Vor allem wichtig, dass man sich dessen bewusst ist, auch wenn man nichts dagegen tun kann

Teil II

APPENDIX

A

DATEIEN

Alle Dateien sind auch digital unter diesem [Link \[Meh25a\]](#) zu finden.

```
{  
  "nodes": [  
    {  
      "name": "root",  
      "type": "Folder",  
      "attributes": {},  
      "children": [  
        {  
          "name": "leaf 0",  
          "type": "File",  
          "attributes": {  
            "rloc": 10,  
            "functions": 5,  
            "mcc": 100  
          }  
        },  
        {  
          "name": "small_parent_1",  
          "type": "Folder",  
          "attributes": {},  
          "children": [  
            {  
              "name": "leaf 1",  
              "type": "File",  
              "attributes": {  
                "rloc": 30,  
                "functions": 100,  
                "mcc": 100  
              }  
            },  
            {  
              "name": "small_parent_2",  
              "type": "Folder",  
              "attributes": {},  
              "children": [  
                {  
                  "name": "leaf 2",  
                  "type": "File",  
                  "attributes": {  
                    "rloc": 50,  
                    "functions": 150,  
                    "mcc": 100  
                  }  
                }  
              ]  
            ]  
          ]  
        ]  
      ]  
    ]  
  ]  
}
```

```
        "rloc": 30,
        "functions": 10,
        "mcc": 100
    }
},
{
    "name": "small_parent_3",
    "type": "Folder",
    "attributes": {},
    "children": [
        {
            "name": "leaf 3",
            "type": "File",
            "attributes": {
                "rloc": 30,
                "functions": 10,
                "mcc": 100
            }
        }
    ]
},
{
    "name": "huge_parent",
    "type": "Folder",
    "attributes": {},
    "children": [
        {
            "name": "huge leaf",
            "type": "File",
            "attributes": {
                "rloc": 3000,
                "functions": 100,
                "mcc": 100
            }
        }
    ]
}
]
```

Listing A.1: Hndisch erzeugte map

```
{  
  "nodes": [  
    {  
      "name": "root",  
      "type": "Folder",  
      "attributes": {},  
      "children": [  
        {  
          "name": "leaf 0",  
          "type": "File",  
          "attributes": {  
            "rloc": 10,  
            "functions": 5,  
            "mcc": 100  
          }  
        },  
        {  
          "name": "small_parent_1",  
          "type": "Folder",  
          "attributes": {},  
          "children": [  
            {  
              "name": "leaf 1",  
              "type": "File",  
              "attributes": {  
                "rloc": 30,  
                "functions": 100,  
                "mcc": 100  
              }  
            },  
            {  
              "name": "small_parent_2",  
              "type": "Folder",  
              "attributes": {},  
              "children": [  
                {  
                  "name": "leaf 2",  
                  "type": "File",  
                  "attributes": {  
                    "rloc": 30,  
                    "functions": 10,  
                    "mcc": 100  
                  }  
                },  
                {  
                  "name": "small_parent_3",  
                  "type": "Folder",  
                  "attributes": {},  
                  "children": [  
                    {  
                      "name": "leaf 3",  
                      "type": "File",  
                      "attributes": {  
                        "rloc": 30,  
                        "functions": 10,  
                        "mcc": 100  
                      }  
                    }  
                  ]  
                }  
              ]  
            }  
          ]  
        }  
      ]  
    }  
  ]  
}
```

```
"type": "Folder",
"attributes": {},
"children": [
{
  "name": "leaf 3",
  "type": "File",
  "attributes": {
    "rloc": 30,
    "functions": 10,
    "mcc": 100
  }
},
{
  "name": "small_parent_4",
  "type": "Folder",
  "attributes": {},
  "children": [
    {
      "name": "leaf 4",
      "type": "File",
      "attributes": {
        "rloc": 30,
        "functions": 10,
        "mcc": 100
      }
    },
    {
      "name": "small_parent_5",
      "type": "Folder",
      "attributes": {},
      "children": [
        {
          "name": "leaf 5",
          "type": "File",
          "attributes": {
            "rloc": 30,
            "functions": 10,
            "mcc": 100
          }
        },
        {
          "name": "small_parent_6",
          "type": "Folder",
          "attributes": {},
          "children": [
            {

```

```
"name": "leaf 6",
"type": "File",
"attributes": {
    "rloc": 30,
    "functions": 10,
    "mcc": 100
}
},
{
    "name": "small_parent_7",
    "type": "Folder",
    "attributes": {},
    "children": [
        {
            "name": "leaf 7",
            "type": "File",
            "attributes": {
                "rloc": 30,
                "functions": 10,
                "mcc": 100
            }
        },
        {
            "name": "small_parent_8",
            "type": "Folder",
            "attributes": {},
            "children": [
                {
                    "name": "leaf 8",
                    "type": "File",
                    "attributes": {
                        "rloc": 30,
                        "functions": 10,
                        "mcc": 100
                    }
                },
                {
                    "name": "small_parent_9",
                    "type": "Folder",
                    "attributes": {},
                    "children": [
                        {
                            "name": "leaf 9",
                            "type": "File",
                            "attributes": {
```

Listing A.2: Händisch erzeugte map

```
{  
    "link": "https://github.com/nvbn/thefuck",  
    "description": "Magnificent app which corrects your  
        previous console command."  
,  
{  
    "link": "https://github.com/scikit-learn/scikit-learn",  
    "description": "scikit-learn: machine learning in Python"  
,  
{  
    "link": "https://github.com/ansible/ansible",  
    "description": "Ansible is a radically simple IT  
        automation platform that makes your applications and  
        systems easier to deploy and maintain. Automate  
        everything from code deployment to network  
        configuration to cloud management, in a language that  
        approaches plain English, using SSH, with no agents  
        to install on remote systems.  
        https://docs.ansible.com."  
,  
{  
    "link": "https://github.com/CorentinJ/Real-Time-Voice-Cloning",  
    "description": "Clone a voice in 5 seconds to generate  
        arbitrary speech in real-time"  
,  
{  
    "link": "https://github.com/apache/airflow",  
    "description": "Apache Airflow - A platform to  
        programmatically author, schedule, and monitor  
        workflows"  
,  
{  
    "link": "https://github.com/harry0703/MoneyPrinterTurbo",  
    "description": "Generate short videos with one click using  
        AI LLM."  
,  
{  
    "link": "https://github.com/freqtrade/freqtrade",  
    "description": "Free, open source crypto trading bot"  
,  
{  
    "link": "https://github.com/streamlit/streamlit",  
    "description": "Streamlit A faster way to build and share  
        data apps."  
,  
{
```

```
"link":"https://github.com/mlflow/mlflow",
"description":"The open source developer platform to
    build AI/LLM applications and models with confidence.
    Enhance your AI applications with end-to-end
    tracking, observability, and evaluations, all in one
    integrated platform."
},
{
    "link":"https://github.com/commaai/openpilot",
    "description":"openpilot is an operating system for
        robotics. Currently, it upgrades the driver
        assistance system on 300+ supported cars."
},
{
    "link":"https://github.com/pytorch/pytorch",
    "description":"Tensors and Dynamic neural networks in
        Python with strong GPU acceleration"
},
{
    "link":"https://github.com/pytorch/vision",
    "description":"Datasets, Transforms and Models specific
        to Computer Vision"
},
{
    "link":"https://github.com/frappe/erpnext",
    "description":"Free and Open Source Enterprise Resource
        Planning (ERP)"
},
{
    "link":"https://github.com/StevenBlack/hosts",
    "description":"Consolidating and extending hosts files
        from several well-curated sources. Optionally pick
        extensions for porn, social media, and other
        categories."
},
{
    "link":"https://github.com/macrozhang/mall",
    "description":"mallBoot+MyBatis"
},
{
    "link":"https://github.com/zxing/zxing",
    "description":"ZXing (Crossingbarcode scanning library
        for Java, Android"
},
{
    "link":"https://github.com/scwang90/SmartRefreshLayout",
```

```
        "description":"Footer"
    },
{
    "link":"https://github.com/oracle/graal",
    "description":"GraalVM compiles Java applications into
                  native executables that start instantly, scale fast,
                  and use fewer compute resources "
},
{
    "link":"https://github.com/keycloak/keycloak",
    "description":"Open Source Identity and Access Management
                  For Modern Applications and Services"
},
{
    "link":"https://github.com/doocs/leetcode",
    "description":"solutions in any programming language |
                  LeetCodeOffer2 6 "
},
{
    "link":"https://github.com/neo4j/neo4j",
    "description":"Graphs for Everyone"
},
{
    "link":"https://github.com/deeplearning4j/deeplearning4j",
    "description":"Suite of tools for deploying and training
                  deep learning models using the JVM. Highlights
                  include model import for keras, tensorflow, and
                  onnx/pytorch, a modular and tiny c++ library for
                  running math code and a java based math library on
                  top of the core c++ library. Also includes samediff:
                  a pytorch/tensorflow like library for running deep
                  learn..."
},
{
    "link":"https://github.com/OpenAPITools/openapi-generator",
    "description":"OpenAPI Generator allows generation of API
                  client libraries (SDK generation), server stubs,
                  documentation and configuration automatically given
                  an OpenAPI Spec (v2, v3)"
},
{
    "link":"https://github.com/SonarSource/sonarqube",
    "description":"Continuous Inspection"
},
{
    "link":"https://github.com/libgdx/libgdx",
    "description": "The LibGDX project is a Java-based game engine for cross-platform game development. It provides a set of tools and libraries for creating games for desktop, mobile, and console platforms. Key features include a physics engine, audio support, and a 3D rendering API."}
```

```
"description":"Desktop/Android/HTML5/iOS Java game
development framework"
},
{
"link":"https://github.com/flyway/flyway",
"description":"Flyway by Redgate Database Migrations Made
Easy."
},
{
"link":"https://github.com/openjdk/jdk",
"description":"JDK main-line development
https://openjdk.org/projects/jdk"
},
{
"link":"https://github.com/arduino/Arduino",
"description":"Arduino IDE 1.x"
},
{
"link":"https://github.com/sveltejs/svelte",
"description":"web development for the rest of us"
},
{
"link":"https://github.com/danielmiessler/Fabric",
"description":"Fabric is an open-source framework for
augmenting humans using AI. It provides a modular
system for solving specific problems using a
crowdsourced set of AI prompts that can be used
anywhere."
},
{
"link":"https://github.com/open-webui/open-webui",
"description":"User-friendly AI Interface (Supports
Ollama, OpenAI API, ...)"
},
{
"link":"https://github.com/atom/atom",
"description":":atom: The hackable text editor"
},
{
"link":"https://github.com/phaserjs/phaser",
"description":"Phaser is a fun, free and fast 2D game
framework for making HTML5 games for desktop and
mobile web browsers, supporting Canvas and WebGL
rendering."
},
{
```

```
"link":"https://github.com/swagger-api/swagger-ui",
"description":"Swagger UI is a collection of HTML,
    JavaScript, and CSS assets that dynamically generate
    beautiful documentation from a Swagger-compliant API."
},
{
    "link":"https://github.com/handsontable/handsontable",
    "description":"JavaScript Data Grid / Data Table with a
        Spreadsheet Look & Feel. Works with React, Angular,
        and Vue. Supported by the Handsontable team "
},
{
    "link":"https://github.com/zhaolee/ChromeAppHeroes",
    "description":" ChromePluginHeroes, Write a Chinese
        manual for the excellent Chrome plugin, let the
        Chrome plugin heroes benefit the human~ 01"
},
{
    "link":"https://github.com/aframevr/aframe",
    "description":":a: Web framework for building virtual
        reality experiences."
},
{
    "link":"https://github.com/4ian/GDevelop",
    "description":"Open-source, cross-platform
        2D/3D/multiplayer game engine designed for everyone."
},
{
    "link":"https://github.com/facebook/react",
    "description":"The library for web and native user
        interfaces."
},
{
    "link":"https://github.com/gatsbyjs/gatsby",
    "description":"The best React-based framework with
        performance, scalability and security built in."
},
{
    "link":"https://github.com/nodejs/node",
    "description":"Node.js JavaScript runtime "
},
{
    "link":"https://github.com/ToolJet/ToolJet",
    "description":"Low-code platform for building business
        applications. Connect to databases, cloud storages,
        GraphQL, API endpoints, Airtable, Google sheets,
```

```

        OpenAI, etc and build apps using drag and drop
        application builder. Built using
        JavaScript/TypeScript. "
},
{
  "link":"https://github.com/romkatv/powerlevel10k",
  "description":"A Zsh theme"
},
{
  "link":"https://github.com/testssl/testssl.sh",
  "description":"Testing TLS/SSL encryption anywhere on any
    port"
},
{
  "link":"https://github.com/linux-surface/linux-surface",
  "description":"Linux Kernel for Surface Devices"
},
{
  "link":"https://github.com/ONLYOFFICE/DocumentServer",
  "description":"ONLYOFFICE Docs is a free collaborative
    online office suite comprising viewers and editors
    for texts, spreadsheets and presentations, forms and
    PDF, fully compatible with Office Open XML formats:
    .docx, .xlsx, .pptx and enabling collaborative
    editing in real time."
},
{
  "link":"https://github.com/facebookarchive/caffe2",
  "description":"Caffe2 is a lightweight, modular, and
    scalable deep learning framework."
},
{
  "link":"https://github.com/mitchellkrogza/
    nginx-ultimate-bad-bot-blocker",
  "description":"Nginx Block Bad Bots, Spam Referrer
    Blocker, Vulnerability Scanners, User-Agents,
    Malware, Adware, Ransomware, Malicious Sites, with
    anti-DDOS, Wordpress Theme Detector Blocking and
    Fail2Ban Jail for Repeat Offenders"
},
{
  "link":"https://github.com/dtcooper/raspotify",
  "description":"A Spotify Connect client that mostly Just
    Works"
},
{

```

```
"link":"https://github.com/armbian/build",
"description":"Armbian Linux build framework generates
custom Debian or Ubuntu image for x86, aarch64,
riscv64 & armhf"
},
{
"link":"https://github.com/filsv/iOSDeviceSupport",
"description":"Xcode iPhoneOS (iOS) DeviceSupport files
(6.0 - 17.0)"
},
{
"link":"https://github.com/prasanthrangan/hyprdots",
"description":="// Aesthetic, dynamic and minimal dots for
Arch hyprland"
},
{
"link":"https://github.com-hoverbike1/T0TK-Mods-collection",
"description":"Mod repo for Tears of The Kingdom (T0TK)
for Switch and Switch Emulation"
},
{
"link":"https://github.com/HyDE-Project/HyDE",
"description":"HyDE, your Development Environment"
},
{
"link":"https://github.com/yonggekkk/x-ui-yg",
"description":"x-uiXhttpargoPsiphonVPN30sing-boxclash-meta"
},
{
"link":"https://github.com/fuzhengwei/CodeGuide",
"description":":books: Java Java()"
},
{
"link":"https://github.com/pbatard/rufus",
"description":"The Reliable USB Formatting Utility"
},
{
"link":"https://github.com/mpv-player/mpv",
"description":"Command line media player"
},
{
"link":"https://github.com/jart/cosmopolitan",
"description":"build-once run-anywhere c library"
},
{
"link":"https://github.com/coolsnowwolf/lede",
```

```
"description":"Lean's LEDE source"
},
{
  "link":"https://github.com/wazuh/wazuh",
  "description":"Wazuh - The Open Source Security Platform.
    Unified XDR and SIEM protection for endpoints and
    cloud workloads."
},
{
  "link":"https://github.com/videolan/vlc",
  "description":"VLC media player - All pull requests are
    ignored, please use MRs on
    https://code.videolan.org/videolan/vlc"
},
{
  "link":"https://github.com/reactos/reactos",
  "description":"A free Windows-compatible Operating System"
},
{
  "link":"https://github.com/TelegramMessenger/Telegram-iOS",
  "description":"Telegram-iOS"
},
{
  "link":"https://github.com/yugabyte/yugabyte-db",
  "description":"YugabyteDB - the cloud native distributed
    SQL database for mission-critical applications."
},
{
  "link":"https://github.com/OnionUI/Onion",
  "description":"OS overhaul for Miyoo Mini and Mini+"
},
{
  "link":"https://github.com/wine-mirror/wine",
  "description":null
},
{
  "link":"https://github.com/wireshark/wireshark",
  "description":"Read-only mirror of Wireshark's Git
    repository at https://gitlab.com/wireshark/wireshark.
    GitHub won't let us disable pull requests. THEY WILL
    BE IGNORED HERE Upload them at GitLab instead."
},
{
  "link":"https://github.com/darktable-org/darktable",
  "description":"darktable is an open source photography
    workflow application and raw developer"
```

```
},
{
  "link": "https://github.com/hanwckf/rt-n56u",
  "description": "Padavan"
}
]
```

Listing A.3: Die zur Analyse ausgewählten GitHub-Repositories

LITERATUR

- [Sofa] [Online; besucht 17-Juni-2025]. 2019. URL: <https://www.studysmarter.de/studium/informatik-studium/softwareentwicklung/softwaremetriken/>.
- [Sofb] [Online; besucht 22-Juni-2025]. 2022. URL: <https://octoverse.github.com/2022/state-of-open-source>.
- [Iso] 2024. URL: <https://www.iso25000.com/index.php/en/iso-2500-standards/iso-25010>.
- [Wor] [Online; besucht 17-Juni-2025]. 2025. URL: <https://www.gitclear.com/four%5Fworst%5Fsoftware%5Fmetrics%5Fagitating%5Fdevelopers>.
- [D3ta] [Online; besucht 19-Mai-2025]. 2025. URL: <https://d3js.org/d3-hierarchy/treemap>.
- [D3tb] [Online; besucht 27-Juli-2025]. 2025. URL: <https://github.com/d3/d3-hierarchy/blob/main/src/treemap/squarify.js>.
- [Iee] [Online; besucht 29-Juli-2025]. 2025. URL: <https://ieeexplore.ieee.org/Xplorehelp/searching-ieee-xplore/search-examples>.
- [AO+19] Islam Al Omari, Razan Al Omoush, Haneen Innab und A El-hassan. "Visualizing Program Quality-A Topological Taxonomy of Features". In: *2019 2nd International Conference on new Trends in Computing Sciences (ICTCS)*. IEEE. 2019, S. 1–10.
- [ADo7] Sazzadul Alam und Philippe Dugerdil. "EvoSpaces: 3D Visualization of Software Architecture." In: *SEKE*. Bd. 7. Citeseer. 2007, S. 500.
- [Atz+21] Daniel Atzberger, Tim Cech, Merlin Haye, Maximilian Söchting, Willy Scheibel, Daniel Limberger und Jürgen Döllner. "Software Forest: A Visualization of Semantic Similarities in Source Code using a Tree Metaphor". In: Feb. 2021, S. 112–122. DOI: [10.5220/0010267601120122](https://doi.org/10.5220/0010267601120122).
- [Bal+16] Gergo Balogh, Tamás Gergely, Árpád Beszédes und Tibor Gyimóthy. "Using the City Metaphor for Visualizing Test-Related Metrics". In: *2016 IEEE 23rd International Conference on Software Analysis, Evolution, and Reengineering (SANER)*. Bd. 2. 2016, S. 17–20. DOI: [10.1109/SANER.2016.48](https://doi.org/10.1109/SANER.2016.48).
- [BD05] Michael Balzer und Oliver Deussen. "Exploring relations within software systems using treemap enhanced hierarchical graphs". In: *3rd IEEE international workshop on visualizing software for understanding and analysis*. IEEE. 2005, S. 1–6.

- [BDLo5] Michael Balzer, Oliver Deussen und Claus Lewerentz. "Voronoi treemaps for the visualization of software metrics". In: *Proceedings of the 2005 ACM symposium on Software visualization*. 2005, S. 165–172.
- [BK01] S. Bassil und R.K. Keller. "Software visualization tools: survey and analysis". In: *Proceedings 9th International Workshop on Program Comprehension. IWPC 2001*. 2001, S. 7–17. DOI: [10.1109/WP.C.2001.921708](https://doi.org/10.1109/WP.C.2001.921708).
- [BHVWoo] Mark Bruls, Kees Huizing und Jarke J Van Wijk. "Squarified treemaps". In: *Data Visualization 2000: Proceedings of the Joint EUROGRAPHICS and IEEE TCVG Symposium on Visualization in Amsterdam, The Netherlands*. Springer. 2000, S. 33–42.
- [Bur15] Michael Burch. "Visualizing software metrics in a software system hierarchy". In: *International Symposium on Visual Computing*. Springer. 2015, S. 733–744.
- [CAC20] Andrea Capiluppi, Nemitari Ajienka und Steve Counsell. "The effect of multiple developers on structural attributes: A Study based on java software". In: *Journal of Systems and Software* 167 (2020), S. 110593. ISSN: 0164-1212. DOI: <https://doi.org/10.1016/j.jss.2020.110593>. URL: <https://www.sciencedirect.com/science/article/pii/S016412122030073X>.
- [CZ10] Pierre Caserta und Olivier Zendra. "Visualization of the static aspects of software: A survey". In: *IEEE transactions on visualization and computer graphics* 17.7 (2010), S. 913–933.
- [Cru+16] Adriana Cruz, Camila Bastos, Paulo Afonso und Heitor Costa. "Software visualization tools and techniques: A systematic review of the literature". In: *2016 35th International Conference of the Chilean Computer Science Society (SCCC)*. IEEE. 2016, S. 1–12.
- [EH98] Institute of Electrical und Electronics Engineers (Hrsg.) *IEEE Std 1061-1998: IEEE Standard for a Software Quality Metrics Methodology*. Kapitel 2. Definitions, S. 2. New York: IEEE, 1998. ISBN: 1-55937-529-9.
- [ETB11] Ural Erdemir, Umut Tekin und Feza Buzluca. "E-Quality: A graph based object oriented software quality visualization tool". In: *2011 6th International Workshop on Visualizing Software for Understanding and Analysis (VISSOFT)*. 2011, S. 1–8. DOI: [10.1109/VISSOFT.2011.6069454](https://doi.org/10.1109/VISSOFT.2011.6069454).
- [GHMo8] Keith Gallagher, Andrew Hatch und Malcolm Munro. "Software architecture visualization: An evaluation framework and its application". In: *IEEE Transactions on Software Engineering* 34.2 (2008), S. 260–270.

- [Gen+21] Patric Genfer, Johann Grabner, Christina Zoffi, Mario Bernhart und Thomas Grechenig. "Visualizing Metric Trends for Software Portfolio Quality Management". In: *2021 Working Conference on Software Visualization (VISSOFT)*. 2021, S. 88–99. doi: [10.1109/VISSOFT52517.2021.00018](https://doi.org/10.1109/VISSOFT52517.2021.00018).
- [Gmba] Generative AI Seerene GmbH. *Sereene Website*. [Online; besucht 29-Juli-2025]. URL: <https://www.seerene.com/de/>.
- [Gmbb] MaibornWolff GmbH. *CodeCharta GitHub*. [Online; besucht 04-August-2025]. URL: <https://github.com/MaibornWolff/codecharta>.
- [Gmbc] MaibornWolff GmbH. *CodeCharta Web Demo*. [Online; besucht 29-Juli-2025]. URL: <https://codecharta.com/visualization/app/index.html>.
- [Gmbd] MaibornWolff GmbH. *CodeCharta Wiki Analysis*. [Online; besucht 31-Juli-2025]. URL: <https://codecharta.com/docs/analysis/codecharta-shell>.
- [Gor+18] Jochen Gortler, Christoph Schulz, Daniel Weiskopf und Oliver Deussen. "Bubble Treemaps for Uncertainty Visualization". In: *IEEE Transactions on Visualization and Computer Graphics* 24.1 (2018), S. 719–728. doi: [10.1109/TVCG.2017.2743959](https://doi.org/10.1109/TVCG.2017.2743959).
- [GYBo4] Hamish Graham, Hong Yul Yang und Rebecca Berrigan. "A solar system metaphor for 3D visualisation of object oriented software metrics". In: *Proceedings of the 2004 Australasian Symposium on Information Visualisation-Volume 35*. 2004, S. 53–59.
- [HF94] Tracy Hall und Norman Fenton. "Implementing software metrics—the critical success factors". In: *Software Quality Journal* 3 (1994), S. 195–208.
- [HH17] Rinse van Hees und Jurriaan Hage. "Stable and predictable Voronoi treemaps for software quality monitoring". In: *Information and Software Technology* 87 (2017), S. 242–258.
- [HVW05] Danny Holten, Roel Vliegen und Jarke J Van Wijk. "Visual realism for the visualization of software metrics". In: *3rd IEEE International Workshop on Visualizing Software for Understanding and Analysis*. IEEE. 2005, S. 1–6.
- [Hor+15] Jeffery S Horsburgh, Stephanie L Reeder, Amber Spackman Jones und Jacob Meline. "Open source software for visualization and quality control of continuous hydrologic and water quality sensor data". In: *Environmental Modelling & Software* 70 (2015), S. 32–44.
- [HB19] Yuriy Hrytsiuk und Orest Bilas. "Visualization of Software Quality Expert Assessment". In: *2019 IEEE 14th International Conference on Computer Sciences and Information Technologies (CSIT)*. Bd. 2. 2019, S. 156–160. doi: [10.1109/STC-CSIT.2019.8929769](https://doi.org/10.1109/STC-CSIT.2019.8929769).

- [JS91] Brian Johnson und Ben Shneiderman. *Tree-maps: A space filling approach to the visualization of hierarchical information structures*. Techn. Ber. UM Computer Science Department; CS-TR-2657, 1991.
- [Kha+12] Taimur Khan, Henning Barthel, Achim Ebert und Peter Liggesmeyer. "Visualization and evolution of software architectures". In: *Visualization of Large and Unstructured Data Sets: Applications in Geospatial Planning, Modeling and Engineering-Proceedings of IRTG 1131 Workshop 2011*. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik. 2012, S. 25–42.
- [Kha+13] Taimur Khan, Henning Barthel, Achim Ebert und Peter Liggesmeyer. "eCITY: A Tool to Track Software Structural Changes Using an Evolving City". In: *2013 IEEE International Conference on Software Maintenance*. 2013, S. 492–495. DOI: [10.1109/ICSM.2013.80](https://doi.org/10.1109/ICSM.2013.80).
- [Kha+15] Taimur Khan, Henning Barthel, Achim Ebert und Peter Liggesmeyer. "Visual analytics of software structure and metrics". In: *2015 IEEE 3rd Working Conference on Software Visualization (ViSSOFT)*. 2015, S. 16–25. DOI: [10.1109/VISSOFT.2015.7332411](https://doi.org/10.1109/VISSOFT.2015.7332411).
- [Kit+04] Barbara Kitchenham u. a. "Procedures for performing systematic reviews". In: *Keele, UK, Keele University 33.2004* (2004), S. 1–26.
- [KMoo] C. Knight und M. Munro. "Virtual but visible software". In: *2000 IEEE Conference on Information Visualization. An International Conference on Computer Visualization and Graphics*. 2000, S. 198–205. DOI: [10.1109/IV.2000.859756](https://doi.org/10.1109/IV.2000.859756).
- [KHA10] Nicholas Kong, Jeffrey Heer und Maneesh Agrawala. "Perceptual Guidelines for Creating Rectangular Treemaps". In: *IEEE Trans. Visualization & Comp. Graphics (Proc. InfoVis)* (2010). URL: <http://vis.stanford.edu/papers/perception-treemaps>.
- [Kuh+10] Adrian Kuhn, David Erni, Peter Loretan und Oscar Nierstrasz. "Software cartography: Thematic software visualization with consistent layout". In: *Journal of Software Maintenance and Evolution: Research and Practice 22.3* (2010), S. 191–210.
- [LSP05] Guillaume Langelier, Houari Sahraoui und Pierre Poulin. "Visualization-based analysis of quality for large-scale software systems". In: *Proceedings of the 20th IEEE/ACM international Conference on Automated software engineering*. 2005, S. 214–223.
- [LD02] Michele Lanza und Stéphane Ducasse. "Understanding software evolution using a combination of software visualization and software metrics". In: *LMO'02: Langages et Modèles à Objet*. 2002.

- [LS02] C. Lewerentz und F. Simon. "Metrics-based 3D visualization of large object-oriented programs". In: *Proceedings First International Workshop on Visualizing Software for Understanding and Analysis*. 2002, S. 70–77. DOI: [10.1109/VISSOF.2002.1019796](https://doi.org/10.1109/VISSOF.2002.1019796).
- [LFo08] Hao Lü und James Fogarty. "Cascaded treemaps: examining the visibility and stability of structure in treemaps". In: *Proceedings of graphics interface 2008*. 2008, S. 259–266.
- [Lu+17] Liangfu Lu, Shiliang Fan, Maolin Huang, Weidong Huang und Ruolan Yang. "Golden Rectangle Treemap". In: *Journal of Physics: Conference Series* 787.1 (Jan. 2017), S. 012007. DOI: [10.1088/1742-6596/787/1/012007](https://doi.org/10.1088/1742-6596/787/1/012007). URL: <https://dx.doi.org/10.1088/1742-6596/787/1/012007>.
- [Lud] Jochen Ludewig. *Wie gut ist die Software?* URL: <https://elib.uni-stuttgart.de/server/api/core/bitstreams/d40bec3f-ba78-495f-9c44-4e54db9daa30/content>.
- [MFM03] Andrian Marcus, Louis Feng und Jonathan I Maletic. "3D representations for software visualization". In: *Proceedings of the 2003 ACM symposium on Software visualization*. SoftVis '03. San Diego, California: Association for Computing Machinery, 2003, 27–ff. ISBN: 1581136420. DOI: [10.1145/774833.774837](https://doi.org/10.1145/774833.774837). URL: <https://doi.org/10.1145/774833.774837>.
- [Meh25a] Benedikt Mehl. *Anhang zu Masterarbeit*. 2025. URL: <https://gitlab.com/BenediktMehl/master-thesis/tree/main/anhang>.
- [Meh25b] Benedikt Mehl. *Squarify Algorithm Example Video*. 2025. URL: <https://github.com/BenediktMehl/master-thesis/blob/main/images/squarify\%5Fexample.gif>.
- [Mer+18] L. Merino, M. Ghafari, C. Anslow und O. Nierstrasz. "A systematic literature review of software visualization evaluation". In: *Journal of Systems and Software* 144 (Juni 2018), S. 165–180. ISSN: 0164-1212. DOI: <https://doi.org/10.1016/j.jss.2018.06.027>. URL: <https://www.sciencedirect.com/science/article/pii/S0164121218301237>.
- [Mer+17] Leonel Merino, Mohammad Ghafari, Craig Anslow und Oscar Nierstrasz. "CityVR: Gameful software visualization". In: *2017 IEEE International conference on software maintenance and evolution (ICSME)*. IEEE. 2017, S. 633–637.
- [MD09] Parastoo Mohagheghi und Vegard Dehlen. "Existing model metrics and relations to model quality". In: *2009 ICSE Workshop on Software Quality*. IEEE. 2009, S. 39–45.
- [PRW03] Michael J. Pacione, Marc Roper und Murray Wood. "A comparative evaluation of dynamic visualisation tools". English. In: *Proceedings of the 10th Working Conference on Reverse Engineering (WCW 2003)*. Hrsg. von M. Lanza, S. Ducasse und M. Pinzger. Requires Template change to Chapter in Book/Report/-

- Conference proceeding : Proceedings of the 10th Working Conference on Reverse Engineering (WCRE 2003); 10th Working Conference on Reverse Engineering, WCRE ; Conference date: 13-11-2003 Through 16-11-2003. 2003, S. 80–89. doi: [10.1109/WCRE.2003.1287239](https://doi.org/10.1109/WCRE.2003.1287239).
- [PSE04] J.W. Paulson, G. Succi und A. Eberlein. "An empirical study of open-source and closed-source software products". In: *IEEE Transactions on Software Engineering* 30.4 (2004), S. 246–256. doi: [10.1109/TSE.2004.1274044](https://doi.org/10.1109/TSE.2004.1274044).
- [Rag+05] S. Raghunathan, A. Prasad, B.K. Mishra und Hsihui Chang. "Open source versus closed source: software quality in monopoly and competitive markets". In: *IEEE Transactions on Systems, Man, and Cybernetics - Part A: Systems and Humans* 35.6 (2005), S. 903–918. doi: [10.1109/TSMCA.2005.853493](https://doi.org/10.1109/TSMCA.2005.853493).
- [Rei95] Steven P. Reiss. "An Engine for the 3D Visualization of Program Information". In: *Journal of Visual Languages & Computing* 6.3 (1995), S. 299–323. ISSN: 1045-926X. doi: <https://doi.org/10.1006/jvlc.1995.1017>. URL: <https://www.sciencedirect.com/science/article/pii/S1045926X85710178>.
- [Saj+14] Hitesh Sajnani, Vaibhav Saini, Joel Ossher und Cristina V. Lopes. "Is Popularity a Measure of Quality? An Analysis of Maven Components". In: *2014 IEEE International Conference on Software Maintenance and Evolution. 2014*, S. 231–240. doi: [10.1109/ICSM.2014.45](https://doi.org/10.1109/ICSM.2014.45).
- [SLD20] Willy Scheibel, Daniel Limberger und Jürgen Döllner. "Survey of treemap layout algorithms". In: *Proceedings of the 13th international symposium on visual information communication and interaction. 2020*, S. 1–9.
- [Sch19] Gast Carina Schmitz. *Wenn Software auf Qualität (s-Metriken) trifft – Teil 2 - MEDtech Ingenieur GmbH*. [Online; besucht 17-Juni-2025]. Jan. 2019. URL: <https://medtech-ingeneur.de/wenn-software-auf-qualitaet-s-metriken-trifft-teil-2/>.
- [Sch23] Google Scholar. *Academia Stack Exchange*. [Online; besucht 29-Juli-2025]. Apr. 2023. URL: <https://academia.stackexchange.com/questions/195737/google-scholar-mixes-all-or-operators-to-one-single-or-condition>.
- [TCO8] Alfredo R Teyseyre und Marcelo R Campo. "An overview of 3D software visualization". In: *IEEE transactions on visualization and computer graphics* 15.1 (2008), S. 87–105.
- [TKS20] Vishvajeet Thakur, Marouane Kessentini und Tushar Sharma. "QScored: An Open Platform for Code Quality Ranking and Visualization". In: *2020 IEEE International Conference on Software Maintenance and Evolution (ICSM)*. 2020, S. 818–821. doi: [10.1109/ICSM46990.2020.00101](https://doi.org/10.1109/ICSM46990.2020.00101).

- [VWW99] J.J. Van Wijk und H. Van de Wetering. "Cushion treemaps: visualization of hierarchical information". In: *Proceedings 1999 IEEE Symposium on Information Visualization (InfoVis'99)*. 1999, S. 73–78. DOI: [10.1109/INFVIS.1999.801860](https://doi.org/10.1109/INFVIS.1999.801860).
- [VCT18] Eduardo Faccin Vernier, João Luiz Dihl Comba und Alexandru Cristian Telea. "A stable greedy insertion treemap algorithm for software evolution visualization". In: *2018 31st SIBGRAPI Conference on Graphics, Patterns and Images (SIBGRAPI)*. IEEE. 2018, S. 158–165.
- [VK17] Jeffrey Voas und Rick Kuhn. "What Happened to Software Metrics?" In: *Computer* 50.5 (Mai 2017), 88–98. DOI: <https://doi.org/10.1109/mc.2017.144>. URL: <https://tsapps.nist.gov/publication/get%5Fpdf.cfm?pub%5Fid=922615>.
- [WLo7] Richard Wettel und Michele Lanza. "Visualizing Software Systems as Cities". In: *2007 4th IEEE International Workshop on Visualizing Software for Understanding and Analysis*. 2007, S. 92–99. DOI: [10.1109/VISSOF.2007.4290706](https://doi.org/10.1109/VISSOF.2007.4290706).
- [WPo4] Autoren der Wikimedia-Projekte. *Beschreibung von Eigenschaften einer Software durch Zahlen*. [Online; besucht 17-Juni-2025]. Feb. 2004. URL: <https://de.wikipedia.org/wiki/Softwaremetrik>.
- [Wit18] Frank Witte. "Metriken für die Softwarequalität". In: *Metriken für das Testreporting: Analyse und Reporting für wirkungsvolles Testmanagement*. Wiesbaden: Springer Fachmedien Wiesbaden, 2018, S. 63–70. ISBN: 978-3-658-19845-9. DOI: [10.1007/978-3-658-19845-9_8](https://doi.org/10.1007/978-3-658-19845-9_8). URL: <https://doi.org/10.1007/978-3-658-19845-9\%5F8>.
- [YM98] P. Young und M. Munro. "Visualising software in virtual reality". In: *Proceedings. 6th International Workshop on Program Comprehension. IWPC'98 (Cat. No.98TB100242)*. 1998, S. 19–26. DOI: [10.1109/WPC.1998.693276](https://doi.org/10.1109/WPC.1998.693276).