INHALTSVERZEICHNIS

Teil I

THESIS

1

EINLEITUNG

1.1 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 Softwaresystem 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; WL07; Rei95; KM00].

Insbesondere 3D-Metaphern, wie die Stadt-Metapher [WL07], 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 [LF08]. 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.

1.2 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.

1.2.1 *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.[Sof]

> 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 vollumfänglich 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 **??**).

1.2.2 *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

1.2.3 *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 feeling is that we are looking at things from 'outside'. 3D visualizations on the other hand provide the potential to create such an immersive experience [WL07, 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 1.1).

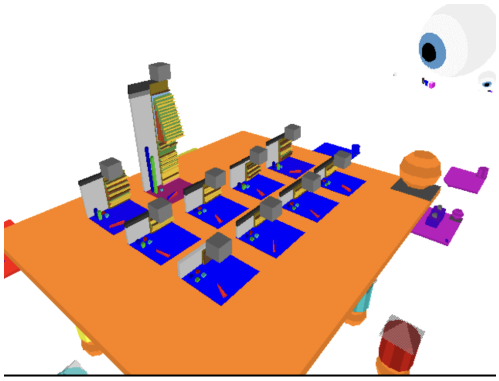


Abbildung 1.1: Beispiel für eine 3D-Visualisierung von Young und Munro [YM98, S. 6]

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 manuelll 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.
  - *Visuelle Metaphern:* Der Einsatz eingängiger Metaphern erleichtert das Verständnis komplexer Softwarestrukturen.
- **Abstraktion:** Unwesentliche Details sollten ausgeblendet werden, um ein verständliches, abstrahiertes Modell zu erzeugen.

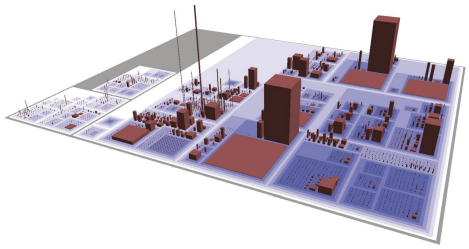


Abbildung 1.2: Beispiel für eine originale CodeCity-Visualisierung [WL07, S. 2]

- **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.

1.2.3.1 *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 [WL07]. 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 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 1.2).

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 vorallem 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 1.2.4). 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 ungenutze *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).

1.2.4 *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 1.3) oder Venn-Diagramme (Abbildung 1.4) konnten diese Anforderungen nicht erfüllen. Sie leiden unter Problemen wie ineffizienter Flächenausnutzung und fehlender Möglichkeit, zusätzlich zu den Beziehungen auch quantitativ-metrische 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 certainly preclude this Venn diagram representation from serious consideration for larger structures.”[JS91, S. 5]

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.

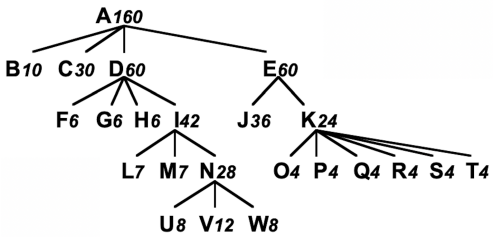


Abbildung 1.3: Beispiel für ein Baumdiagramm

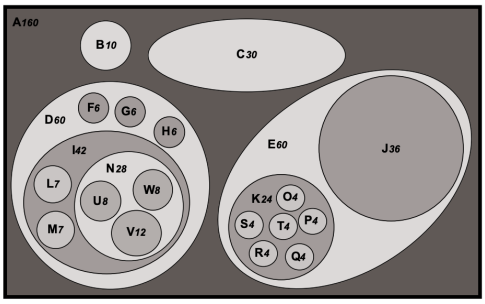


Abbildung 1.4: Beispiel für ein Venn-Diagramm

- 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 1.5 zu sehen.

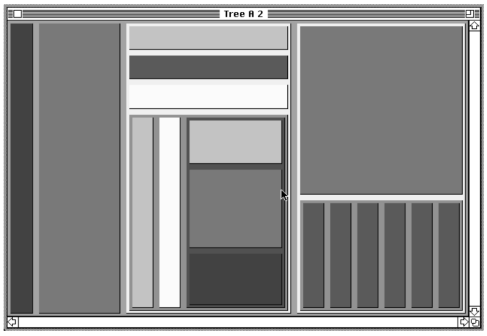


Abbildung 1.5: Beispiel für eine Treemap

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 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 1.2.4.1).

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.

1.2.4.1 *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 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. [BHVWoo, S. 5] erläutert, wobei die Erklärung

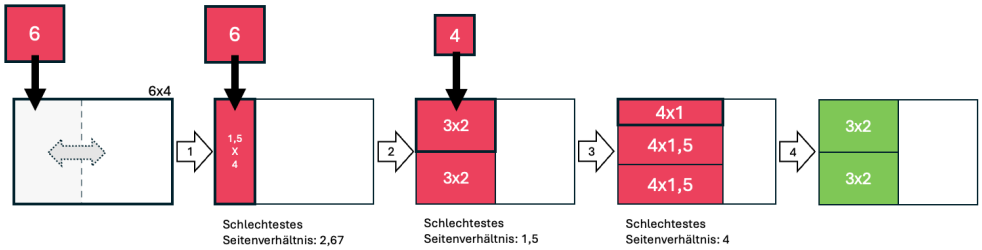


Abbildung 1.6: Beispiel für die erste Reihe des Squarify-Algorithmus. Die Reihe ist im linken großen Rechteck in hellgrau dargestellt, die kann nach rechts in der Breite variieren. Rot sind die Rechtecke, die in ihrer Breite und Höhe noch nicht festgelegt sind. Grün sind die Rechtecke, bei denen die Breite und Höhe neu festgelegt wurde.

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 einsortiert 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 1.6). Diese Reihe wird solange ergänzt, bis das schlechteste Seitenverhältnis (also das ungünstigste der enthaltenen 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 Reige 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 einsortiert (siehe Schritt 4).

Nachdem nun die erste Reihe abgeschlossen ist, wird eine neue Reihe eröffnet (siehe Abbildung 1.7). 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 [Meh25] 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 einsortiert sind (siehe Abbildung 1.8).

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:

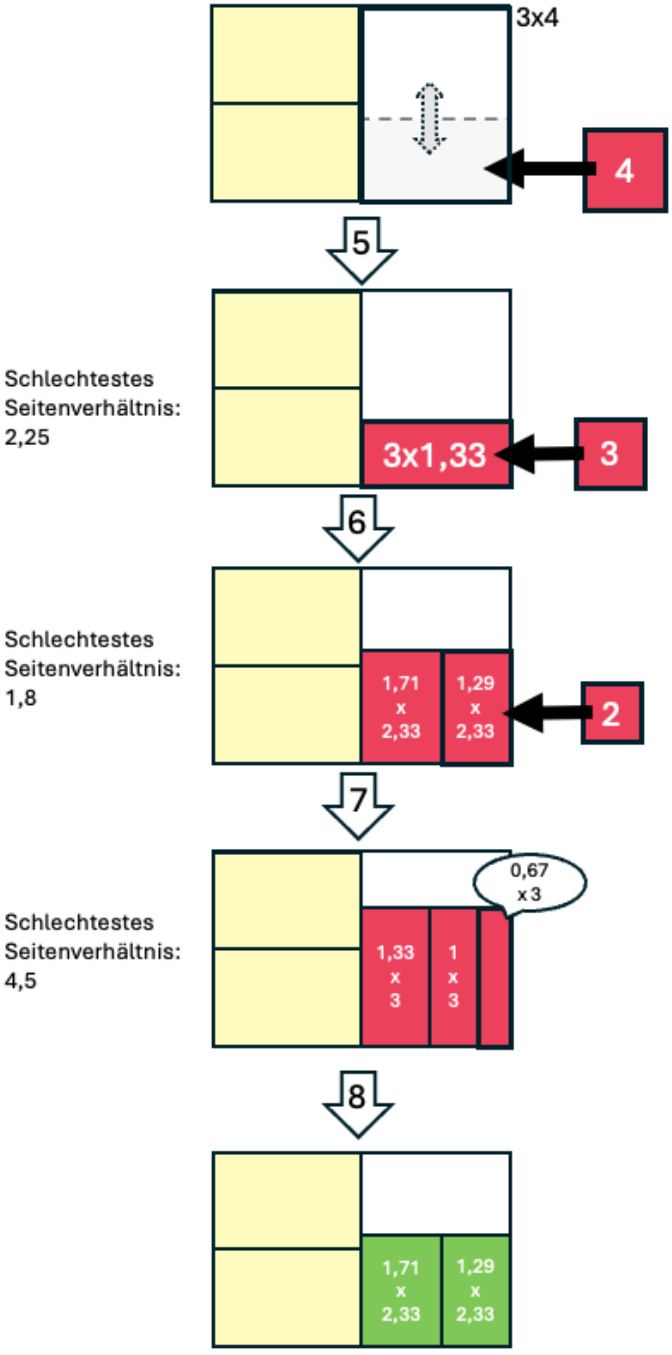
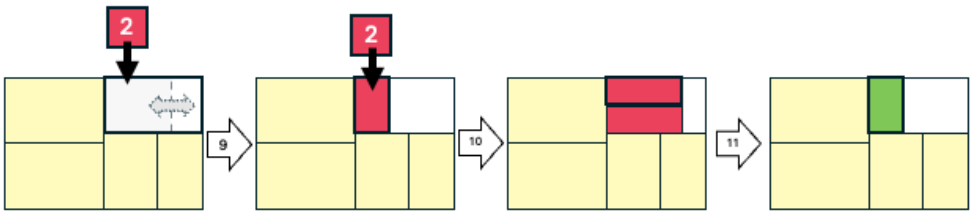
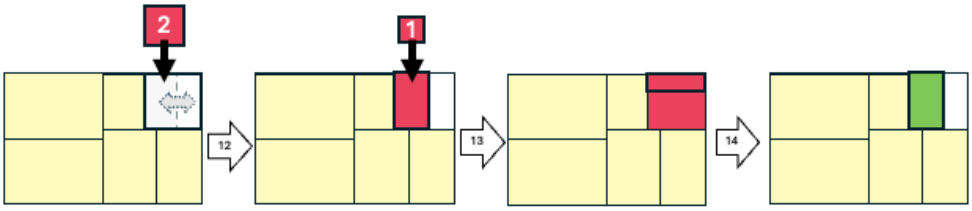


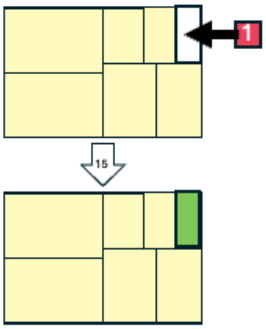
Abbildung 1.7: Beispiel für die zweite Reihe des Squarify-Algorithmus. Die Reihe ist im linken großen Rechteck in hellgrau dargestellt, die kann nach rechts in der Breite variieren. Rot sind die Rechtecke, die in ihrer Breite und Höhe noch nicht festgelegt sind. Gelb sind die Rechtecke die teil von zuvor abgeschlossenen Reihen sind. Grün sind die Rechtecke, bei denen die Breite und Höhe neu festgelegt wurde.



Reihe 3



Reihe 4



Reihe 5

Abbildung 1.8: Beispiel für die letzen 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.

$$\frac{w_i}{l_i} = \frac{w_i \cdot l_i \cdot w^2}{l_i \cdot l_i \cdot w^2} \tag{1.1}$$

$$= \frac{w_i \cdot l_i \cdot w^2}{l_i^2 \cdot \left(\sum_{j=0}^{n} w_j\right)^2} \tag{1.2}$$

$$= \frac{w_i \cdot l_i \cdot w^2}{\left(l_i \cdot \sum_{j=0}^{n} w_j\right)^2} \tag{1.3}$$

$$= \frac{w_i \cdot l_i \cdot w^2}{\left(\sum_{j=0}^{n} l_i \cdot w_j\right)^2} \tag{1.4}$$

$$= \frac{w_i \cdot l_i \cdot w^2}{\left(\sum_{j=0}^{n} l_j \cdot w_j\right)^2} \quad \text{da } \forall i, j \in \{0, \ldots, n\}, l_i = l_j \tag{1.5}$$

$$= \frac{V_i \cdot w^2}{sV^2} \tag{1.6}$$

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 Algotihmus noch besser zu verstehen, hängen wir folgend den code an, so wie wir in 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 d3.js [D3tb].

```
function squarifyNode(parent: SquarifyNode) {
    // nodes sind die einzusortierenden Knoten
    const nodes: SquarifyNode[] = parent.children
    let row: SquarifyRow,
        x0 = parent.x0,
        y0 = parent.y0,
        i = 0,
        j = 0,
        numberOfChildren = nodes.length,
        width: number,
        length: number,
        value = parent.value,
        sumValue: number,
        minValue: number,
        maxValue: number,
        newRatio: number,
        minRatio: number,
        alpha: number,
        beta: number

```

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

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

Listing 1.1: Vereinfachte und kommentierte Grundlage für den Algorithmus der in dieser Arbeit hauptsächlich verwendet wird

Obwohl sich das ursprüngliche Paper von Bruls et al. [BHVW00] 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 1.9 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. [BHVW00]. In Abbildung 1.10 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.

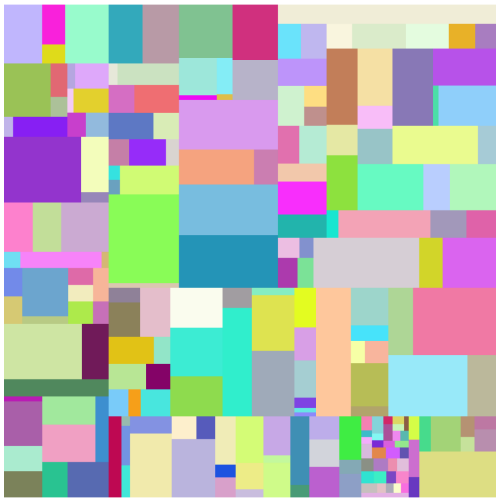


Abbildung 1.9: Beispiel für ein Squarify-Layout mit Annäherung an quadratische Rechtecke (durchschnittliches Seitenverhältnis 1,42)

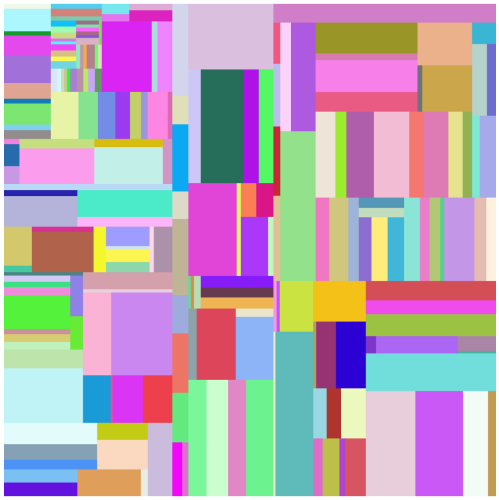


Abbildung 1.10: Beispiel für ein Squarify-Layout mit Annäherung an den Wert 5 (durchschnittliches Seitenverhältnis 2,79)

1.3 PROBLEMSTELLUNG

Die zentrale Herausforderung bei der Entwicklung von Stadtmetaphern zur Softwarevisualisierung besteht in der Festlegung eines geeigneten Layouts für die Visualisierung. Im Abschnitt 1.2.3.1 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?

1.3.1 *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. [MFM03, 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 Visualierung 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 un die qualität von software zu beurteilen [WL07]. 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 1.2). 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 1.2: Schema einer Node

Genauer werden wir in Abschnitt **??** 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 1.2.3.1). 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 1.2.3) 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.

1.3.2 *Das Treemap-Problem*

Durch die nähere Erlärung des Treemap-Algorithmus (siehe Abschnitt 1.2.4) wurde gezeigt, dass klassische Treemap-Algorithmen grundlegende Schwächen auf weisen, 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 [BHVW00, 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 1.11, 1.12 und 1.13 sind verschiedene Treemap-Layouts zu sehen, die mithilfe des Squarify-Algorithmus (siehe Abschnitt 1.2.4.1) 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 **??**).

In Abbildung 1.11 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 1.12 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 1.13, 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 1.14. 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

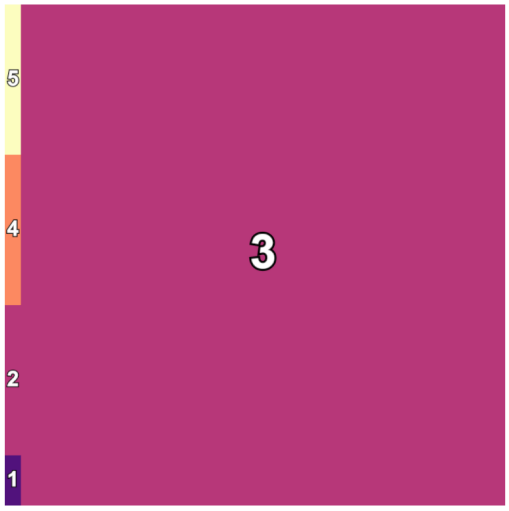


Abbildung 1.11: Treemap-Layout, generiert mit dem Squarify-Algorithmus (Abstand 0).

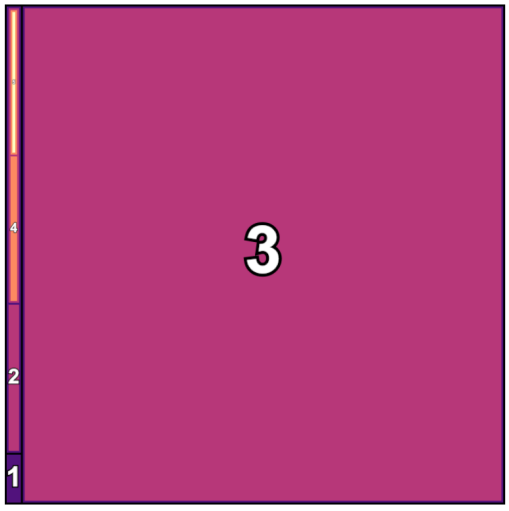


Abbildung 1.12: Treemap-Layout, generiert mit dem Squarify-Algorithmus (Abstand 5).

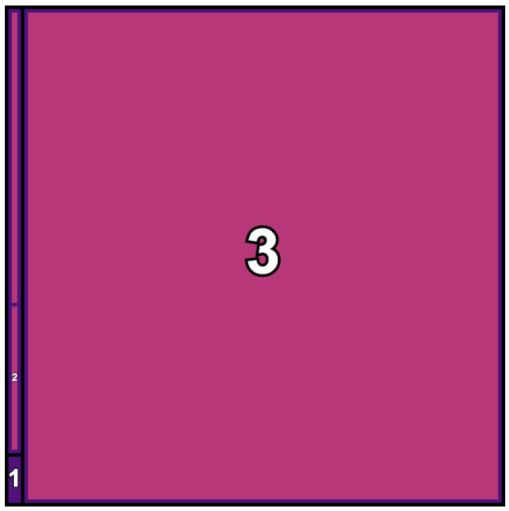


Abbildung 1.13: Treemap-Layout, generiert mit dem Squarify-Algorithmus (Abstand 10).

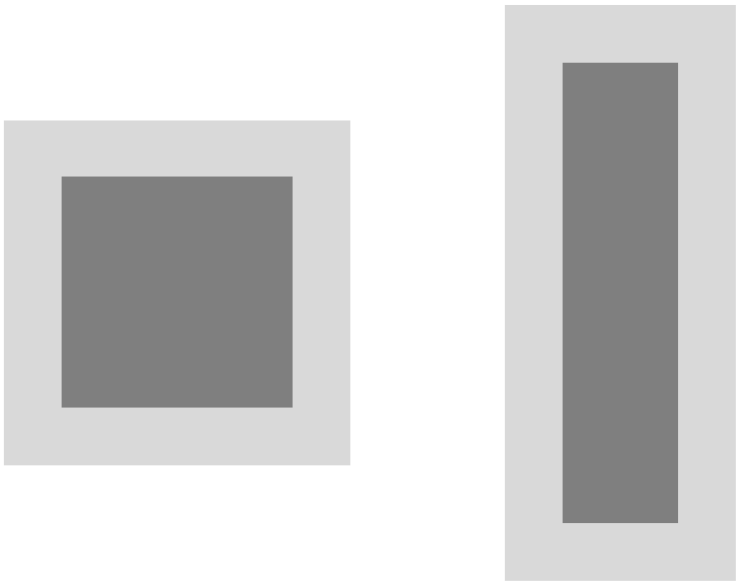


Abbildung 1.14: Zwei Rechtecke der Fläche 16 (dunkelgrau) mit umgebendem Abstand 1 (hellgrau): Links (4×4) mit Gesamtfläche 25 (5×5), rechts (2×8) mit 40 (4×10). Der jeweils notwendige Platz für Abstände hängt vom Seitenverhältnis ab.

Ansätze untersucht werden, um dieses Problem zu lösen (siehe Abschnitt 2.2).

1.4 VERWANDTE ARBEITEN

In diesem Abschnitt werden verwandte Arbeiten vorgestellt, die auch das Layoutproblem betrachten. Außerdem will ich auch konkrete Tools anschauen, die Metriken visualisieren und deren Ansätze betrachten.

Ziel der Suche: ich will 2d layouts finden, die geeignet sein könnten, um Software-Qualitätsmetriken zu visualisieren in 2.5D.

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

Wir wollen möglichst viele Layouts finden, auch welche, die nicht direkt für Software-Qualitätsmetriken gedacht sind oder die nicht direkt für das extrudieren ins drei dimensionale gedacht sind, aber die trotzdem geeignet sein könnten. Deswegen suchen wir auf folgenden Seiten, die 1. nicht wissenschaftlich sind, da wir nach tools suchen und 2. allgemein sind für visualisierungen:

Wissenschaftlich: Google Scholar und IEEE Tools: Google und GitHub

Hierarchical data visualization methods Space-filling Hierarchy Visualization Software Quality Visualization Alternative Tree Structure Visualization context display hierarchy visualizations

Nochmal mehr nach schlagworten schauen Bei cascaded, was für schlagwörter - in welchem kontext wurde es veröffentlicht Software Qualitätsvis mehr einbrigen in die Suche - eher auf diese Niesche eingehen

Methodik auch so beschreiben, wie es wirklich war ... wie wurden die ersten paper gefunden (durch ßufall") um das zu festigen und abzusichern noch eine strukturierte Methodik

1.4.1 *visualization Libraries*

?? Wir schauen uns verschiedene bekannte Open-source Visualisierungsbibliotheken. Oft haben diese eine gute Übersicht über verschiedene Visualisierungen. Der Vorteil ist natürlich dass man diese dann direkt auch verwenden kann, ohne selbst zu implementieren oder man kann den code nehmen und selbst anpassen.

Man muss mit einer gewissen kreativität an die sache heran gehen, es werden alle Visualisierungen betrachtet, es kann natürlich sein ,dass jemand noch mehr finden würde, wir suchen aber hier die geeignetsten heraus

Für die Github suche nutzen wir eine Offene List von visualierungstools [Awe].

wir schauen uns d3.js an, die bekannteste javascript Visualisierungsbibliothek: es gibt extra

Zuerst seaborn in der galery: [Sea] Seaborn bietet keine Visualierungen, die gut für unseren zweck geeignet sein könnten. Das was noch am nächsten kommt, wäre eine heatmap, wie in Abbildung 1.15 zu sehen, die zusätlich

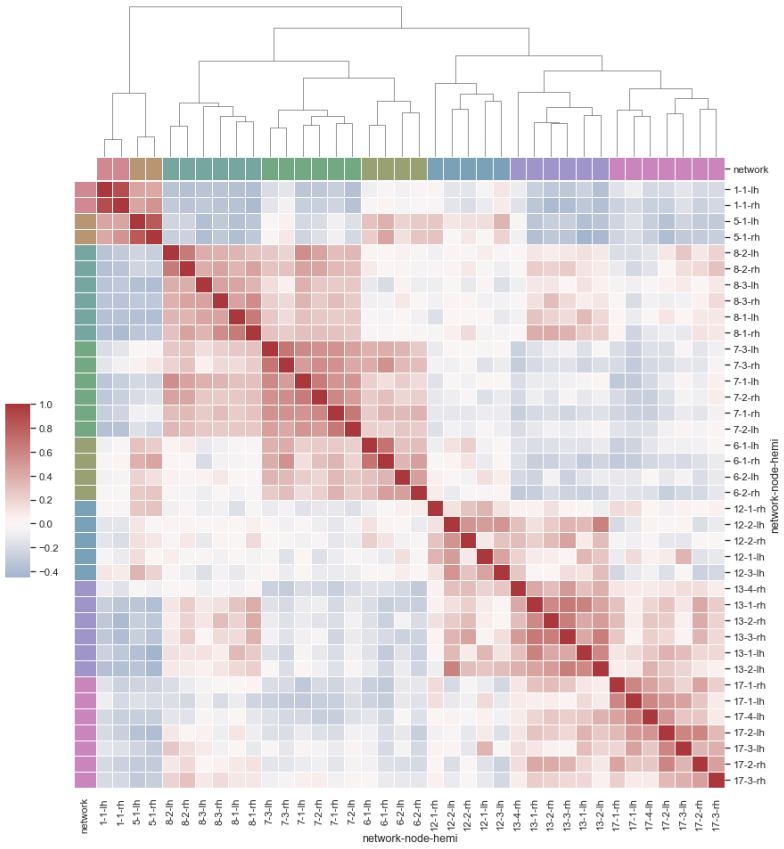


Abbildung 1.15: Beispiel für eine Heatmap aus der Seaborn Galery [Sea].

über ein Baumdiagram informationen über die Struktur darstellt, wie in der Einleitung aber schon gesagt, ist das nicht wirklich geeignet, da es hierbei für große Daten probleme gibt mit baumdigrammen und diese unübersichtlich werden. außerdem sollen die diagramme auch möglichst einfach sein.

1.4.2 *layouts*

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]

1.4.2.1 *Treemap layouts*

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.

Es gibt bisher noch zu wenige ARbeiten, die sich mit dieser Frage beschäftigen. (Oder eigentlich gibt es nur eine die wirklich in die richtung geht - mehr dann bei verwandte arbeiten)

[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 1.3.2). Die Idee ist anders als bei Nested Ansätzen die Kindknoten nicht einfach in den Elternknoten zu zeichnen, sondern sie leicht versetzt *über* dem Elternknoten zu zeichnen (siehe Abbildung 1.16). 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 Schrittigen 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 diser Arbeit aufzuzeigen. Sie beschreiben ihren Schritt wie folgt:

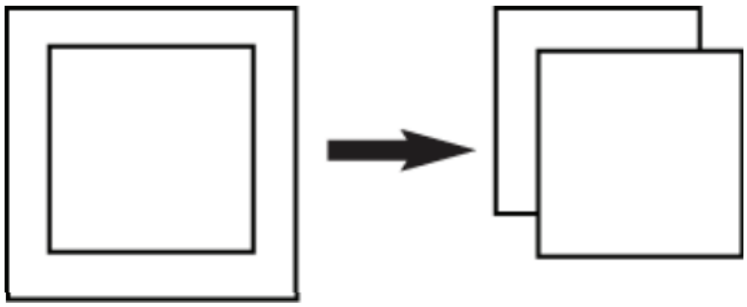


Abbildung 1.16: Links beispielhaf der Nested treemaps Ansatz, bei dem der Kindknoten einfach in dem Elternknoten gezeichnet wird. Rechts der Cascaded Treemap Ansatz, bei dem der Kindknoten als rechteck leicht versetzt nach rechts unten *über* dem Elternknoten gezeichnet wird. Abbildung aus [LFo8, S. 3].

> 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 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 1.17 für deutlich, dass der Algorithmus für den Bereich über von der Kante benauso viel Platz für die Abstände berechnet, wie für den Bereicht unter der Kante, weil eben nur der Platz senkrecht zur Kante betrachtet wird. Und das obwohl offentlich 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 1.3.2), 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

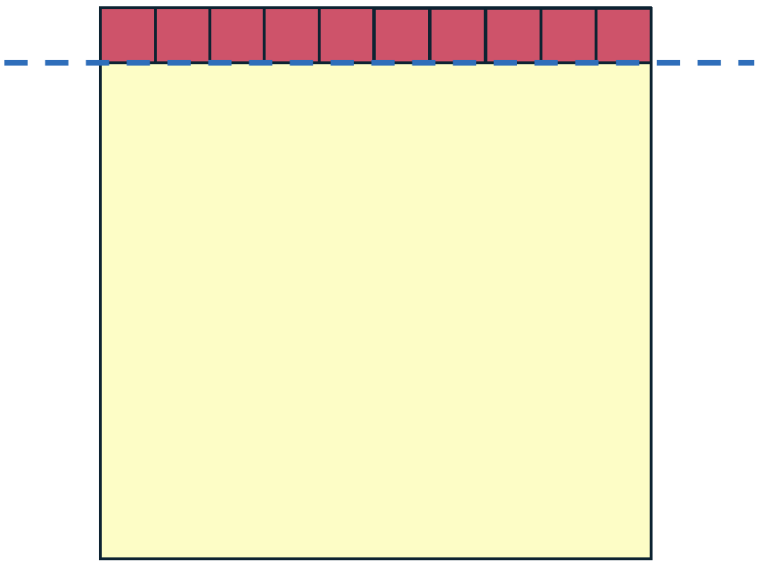


Abbildung 1.17: Beispiel für eine für den Cascaded Treemap Algorithmus schlechte Rechteck-Konstelation

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 Ansatz mit Scaling und festen Knoten zu vergleichen, nur mit dem Umterschied, 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 [LF08, S. 6]).

Die Autoren des ursprungs squarify Algorithmus [BHVW00] stellen in einem anderen Paper [VWW99] eine Idee vor, um Struktur ohne Änderung des Layouts darzustellen undzwar mit Schatten (siehe Abbildung 1.18). Dabei bekommt jeder Knoten, egal ob Eltern- oder Kindknoten, einen Innenrenschatten, wodurch die Struktur der Knoten sichbar 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 **??**).

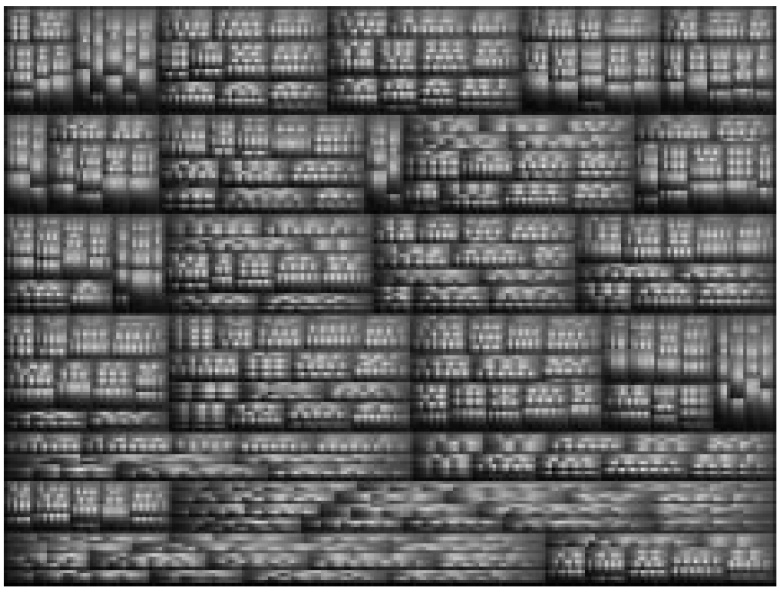


Abbildung 1.18: Beispiel für eine Cushion Treemap [VWW99, S. 4]

Scheibel et al. stellen in ihrem Paper *Survey of treemap layout algorithms*[SLD20] eine Übersicht über die verschiedenen Treemap Layout Algorithmen vor. Sie unterscheiden zwischen den verschiedenen Ansätzen.

Sie kategorieren die verschiedenen Ansätze in 4 Kategorien: Art der Aufteilung, zusätzliche Attribute, Layout Form und Referenzraum Dimension. Uns interresiert hier besonders das zusätzliche Attribut der "Werteünd die Dimension 2D (wie in Abschnitt 1.3 beschrieben). Arten der Aufteilung sind: Packing und splitting. Packing ist dabei die Idee, die Knoten so zu packen, dass sie möglichst wenig Platz verbrauchen und splitting ist die Idee Knoten in kleinere Knoten zu unterteilen. Alle in den Grundlagen (Abschnit 1.2.4) vorgestellten Algorithmen sind also splitting Algorithmen. Sie stellen vier Layout Formen vor: Kreisförmig, rechteckig, konvex und nicht konvex. Alle in den Grundlagen vorgestellten Algorithmen sind rechteckige Layouts. Von den untersuchten 81 Algorithmen sind 54 rechteckige Layouts und 58 splitting Algorithmen. Der in dieser ARbeit speziell untersuchte Treemap Algorithmus passt also genau in die Kategorie der meist verwendeten Ansätze.

1.4.3 *Was suchen wir*

Wir suchen paper oder tools, die metriken oder andere werte/attribute darstellen. Diese Darstellungen sollten entweder space filling sein oder zumindest eine Art von Layout haben, dass die Struktur der Hierarchie darstellt. Die Darstellungen sollten ins drei dimensionale extrahierbar sein. Die Darstellungen werden dann etweder in die kategorie ßquarified treemapöder ändereëingeordnet.

2

HAUPTTEIL

2.1 STADT LAYOUT ANPASSUNGEN

In diesem Abschnitt geht es nicht um die Algorithmen, die zur erzeugung der Layouts verwendet werden, sondern es geht um die Layouts selbst. Hier wird immer konkret begründed, warum ich mich für bestimmte layouts entschieden habe und eventuell auch warum ich mich gegen andere entschieden habe. Dann werden diese Layouts - auf das in dieser Arbeit behandelte konkrete Problem - angepasst, damit sie dann am Ende evaliuiert werden können.

CodeCity: Optionen: Eins: Die erste Änderung ist, dass die Knoten nicht mehr quadratisch sind, um den Platz besser auszunutzen und weniger freie ungenutzte Flächen zu haben. Zwei: Abstände zwischen Knoten könnten auf top level höher sein und nach unten hin immer kleiner werden, um den Platz besser auszunutzen und Probleme mit den Algorithmen zu verkleinern. Drei: Ordner könnten ein Label bekommen, dass die Fläche in eine dimension vergrößert. Anonsten wird das schwer zb. mit schwebenden Labels, da wird alles unübersichtlich. Das vielleicht auch nur auf den Top Level Ordnern?

Ganz Andere Idee: Street maps??? also order werden zu straßen und die Knoten zu Häusern

Ganz Andere Idee: Kreis förmig anordnen? Es gibt ja auch runde städte (oder bar chart)

Ganz Andere Idee: Circle packing

2.2 ERWEITERUNG DES SQUARIFY ALGORITHMUS

In diesem Abschnitt wird der Squarify Algorithmus [BHVWoo], wie er in Abschnitt 1.2.4.1 beschrieben wurde, auf verschiedene Weisen erweitert und angepasst, um das Layoutproblem, wie es in Abschnitt 1.3.2 beschrieben wurde, anzugehen.

2.2.1 *Approximative Fläche*

Die Grundlegende Idee dieser Erweiterung ist es, die Fläche der Knoten plus die Benötigte Fläche für die Abstände vor berechnung des Layouts zu approximieren.

Dafür brauche ich erstmal einen guten Algorithmus der das Layout gut macht, dann kann ich mit KI die Fläche lernen. Ist die frage ob das wirklich so gut funktionieren kann.

Problem man kann sich sehr gut beispiele konstuieren, bei denen das nicth funktionieren wird. Man kann das natürlich mit skalierung wieder lösen, aber das ist natürlich nicht optimal.

2.2.2 *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 Erweitung natürlich nicht optimal funktionieren kann und das auch klar ist, da sich die die Änderung der Größe der Knoten natürlich auch das Layout im Zeweiten Durchlauf ändern wird, wodurch die Größen der Knoten wieder nicht korrekt sind. Was ja überhaupt erst das Grundlegende Problem ist (siehe Abshnictt 1.3). 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 beinhalten, zu verkleinern, um auch den abstand zwischen den Geschwistern herzustellen. Es ist zu erkennen, dass dadurch der Abstand sowohl zwi-

schen 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 Kindernkoten 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 Abhängikeit 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.

2.2.2.1 *Einfache 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.

Algorithm 1 Einfache Größenanpassung

```
1:  function increaseValuesSimple(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
```

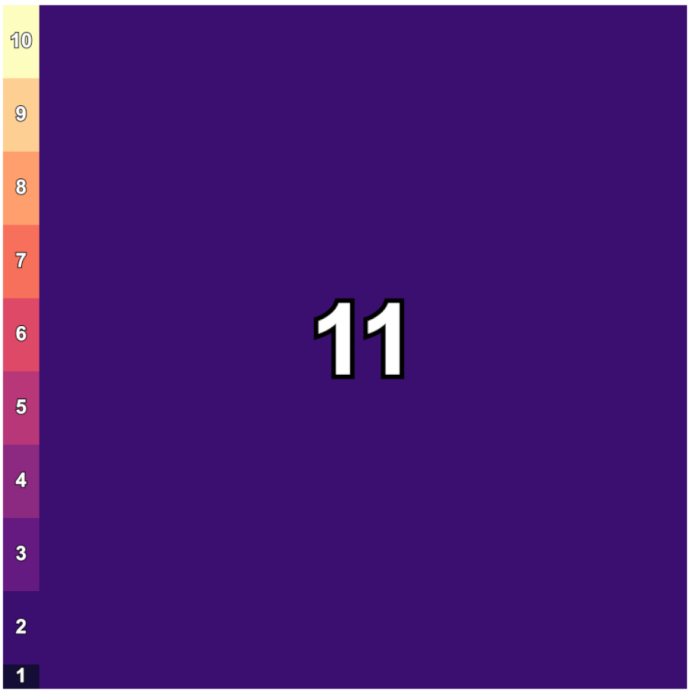


Abbildung 2.1: Treemap Layout generiert mit dem Squarify Algorithmus nach Abschnitt 1.2.4.1 mit einem Abstand von 0 und der einfachen Größenanpassung auf der händisch erstellen map (siehe Anhang).

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 2.1 ist das Layout mit einem Abstand von 0 zu sehen.

In Abbildung 2.2 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 schmaler als im ersten Layout-Schritt. Dadurch wird die Fläche die durch den Abstand eingenommen wird größer als angenommen, weshalb die Fläche des Knotens nach abzug des Abstands im letzen 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 schmaler werden, wodurch die fläche, die der innerere abstand einnimm, 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.

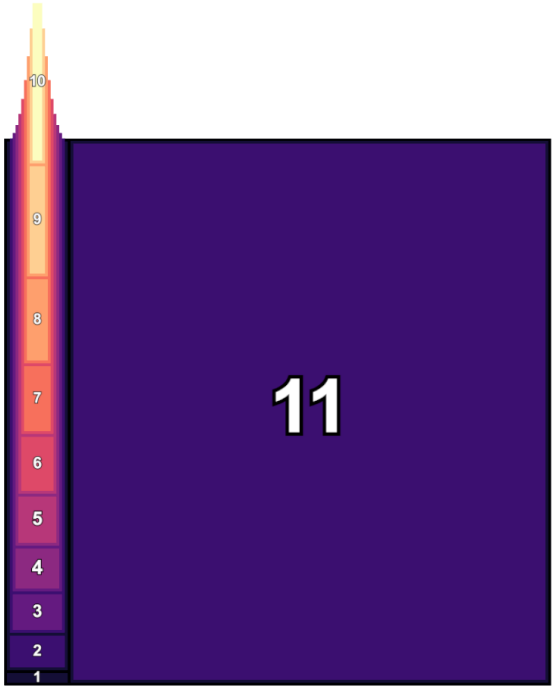


Abbildung 2.2: Treemap Layout generiert mit dem Squarify Algorithmus nach Abschnitt 1.2.4.1 mit einem Abstand von 1 und der einfachen Größenanpassung auf der händisch erstellen map (siehe Anhang).

Dieser Effekt kann einfach behoben werden, wie wir in Abschnitt 2.2.3 sehen werden.

2.2.2.2 *Relative Größenanpassung*

Bei der berechnung zuvor wurde einfach der margin fläche eines Elternknotens berechnet, auf basis der alten Fläche, ohne die Flächenädnerung durch die Kinderknoten zu berücksichtigen. dadurch wird die resultierende Fläche der Elternknoten zu klein sein, da die Fläche der Kinderknoten nicht berücksichtigt wird. Die Idee bei dieser Erweiterung 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 Algorthmus wird im foldenden als Pseudocode dargestellt (siehe Algorithmus 2).

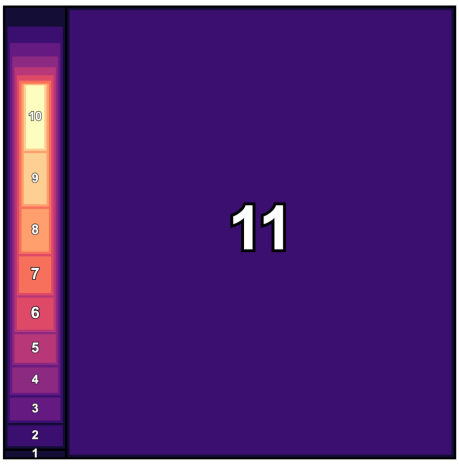


Abbildung 2.3: Treemap Layout generiert mit dem Squarify Algorithmus nach Abschnitt 1.2.4.1 mit einem Abstand von 1 und der relativen Größenanpassung auf der händisch erstellen map (siehe Anhang).

Algorithm 2 Relative Größenanpassung

```
1:  function increaseValues(node: SquarifyNode, margin: number)
2:      childrenValueIncrease ← 0
3:      if node.children then
4:          for child in node.children do
5:              childrenValueIncrease += increaseValues(child, margin)
6:          end for
7:      end if
8:      ratioChildrenValueIncrease ← (node.value + childrenValueIncrease)
    / node.value
9:      valueIncrease ← Math.sqrt(ratioChildrenValueIncrease) * width *
    margin * 2 + Math.sqrt(ratioChildrenValueIncrease) * length * margin
    * 2 + margin * margin * 4 + childrenValueIncrease
10:     node.value += valueIncrease
11:     return valueIncrease
12: 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 2.3 für ein Beispiel.

2.2.3 *Scaling der Knoten*

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

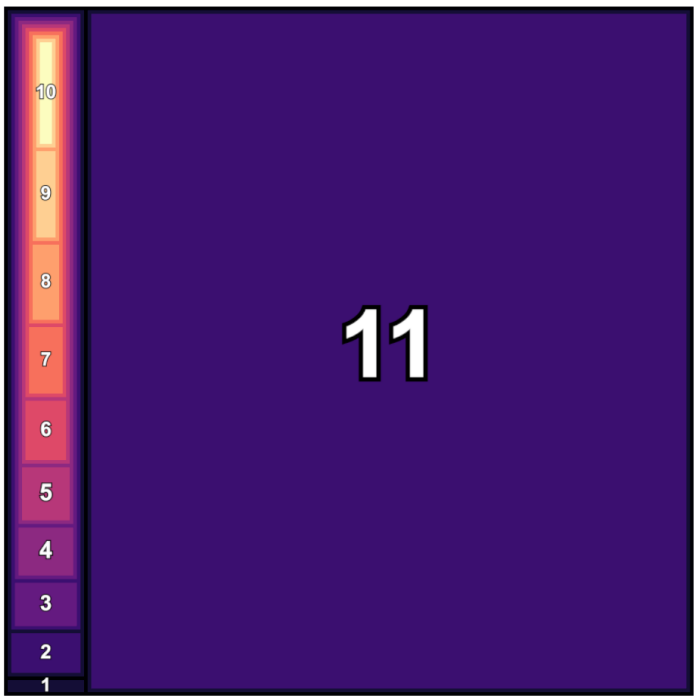


Abbildung 2.4: Treemap Layout generiert mit dem Squarify Algorithmus nach Abschnitt 1.2.4.1 mit einem Abstand von 1 und der einfachen Größenanpassung auf der händisch erstellen map (siehe Anhang) und der Skalierung der Knoten.

dung 2.2 zu sehen ist. Es kann genauso passieren, dass die Fläche innerhalb der Knoten größer wird, wie in Abbildung 2.3 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 2.4 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 2.5, 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

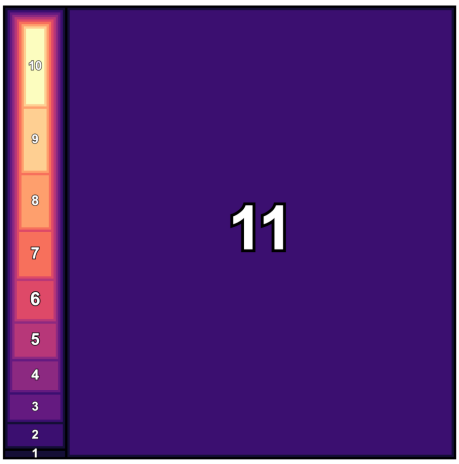


Abbildung 2.5: Treemap Layout generiert mit dem Squarify Algorithmus nach Abschnitt 1.2.4.1 mit einem Abstand von 1 und der relativen Größenanpassung auf der händisch erstellen map (siehe Anhang) und der Skalierung der Knoten.

subjektive Einschätzung wird aber nochmal genauer beläuchtet in der evaluation und vergleich.

2.2.4 *Reihenfolge der Knoten*

Die Reienfolge in der die knoten in das Layout eingefügt werden, hat einen großen Einfluss auf das Layout. [JS91] haben in ihrem Paper herausgefunden, dass die ergebnisse am besten sind, wenn die Knoten in der Reihenfolge der Größe eingefügt werden. Also der Größte zuerst.

2.2.5 *Mehrfache Berechnung*

Die Grundlegende Idee dieser Erweiterung ist es, das Layout mehrfach zu berechnen und dabei die Fläche der Knoten immer weiter anzupassen. Dies wird so lange wiederholt, bis sich die Fläche der Knoten nicht mehr ändert oder eine maximale Anzahl an Iterationen erreicht ist.

2.2.6 *Anpassung der Knoten*

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

2.2.7 *Fazit*

Immernoch straight forward, es gibt aber noch Probleme

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

3

SCHLUSS

3.1 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 visualiert wird. Dann wird verschiedenen unerfahrenen Personen diese verschiedenen Visualierungen 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.

3.2 KRITERIEN FÜR TREEMAP LAYOUTS

Um disese 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.
- **Platznutzung:** Es sollte so wenig Fläche wir 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.
- **Knoten sichtbarkeit:** Es sollte keine Knoten geben, die aufgrund von Abständen oder anderen Gründen nicht sichtbar sind. Dieses Ziel spielt speziell auf das in abschnitt 1.2.4 beschriebene Problem ab.
- **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.
- **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.
- **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)

- **Stabilität:** Die Knoten sollten bei Änderungen die Position und Größe beibehalten, um eine stabile Visualisierung zu gewährleisten.

In dieser Arbeit sollen space filling approaches anaylsiert werden und speziell darauf untersucht werden, wie sie sich eignenen 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 vorallem die splitting algorhtmen 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 algotihmus besonders gut und wann er schlecht ist - hängt natürlich von input daten ab. zb. ab welcher tiefe ist er gut/schlecht. ab wie viel knoten. Ab wie viel knoten unterschied in der größe

3.3 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 übersichlichekeit und einfachheit leiden könnte. Die in diser arbeit untersuchten Ansätze sind alle sehr einfach mit basic formen und daher übersichlich.

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.

Teil II

APPENDIX

LITERATUR

[Sof] [Online; accessed 17-June-2025]. 2019. URL: https://www.studysmarter.de/studium/informatik-studium/softwareentwicklung/softwaremetriken/.

[Iso] 2024. URL: https://www.iso25000.com/index.php/en/iso-25000-standards/iso-25010.

[Sea] [Online; accessed 07-July-2025]. 2024. URL: https://seaborn.pydata.org/examples/index.html.

[Wor] [Online; accessed 17-June-2025]. 2025. URL: https://www.gitclear.com/four\%5Fworst\%5Fsoftware\%5Fmetrics\%5Fagitating\%5Fdevelopers.

[D3ta] [Online; accessed 19-May-2025]. 2025. URL: https://d3js.org/d3-hierarchy/treemap.

[D3tb] [Online; accessed 27-July-2025]. 2025. URL: https://github.com/d3/d3-hierarchy/blob/main/src/treemap/squarify.js.

[Awe] [Online; accessed 07-July-2025]. 2025. URL: https://github.com/hal9ai/awesome-dataviz?tab=readme-ov-file#python-tools.

[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.

[BHVW00] 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.

[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.

[HF94] Tracy Hall und Norman Fenton. "Implementing software metrics–the critical success factors". In: *Software Quality Journal* 3 (1994), S. 195–208.

[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.

[KM00] 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.

[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.

[LF08] 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. 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. URL: https://doi.org/10.1145/774833.774837.

[Meh25] Benedikt Mehl. *Squarify Algorithm Example Video*. [Online; uploaded 27-July-2025]. 2025. URL: https://github.com/BenediktMehl/master-thesis/blob/main/images/squarify_example.gif.

[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.

[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; accessed 17-June-2025]. Jan. 2019. URL: https://medtech-ingenieur.de/wenn-software-auf-qualitaet-s-metriken-trifft-teil-2/.

[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.

[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.

[WL07] 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.

[WP04] Autoren der Wikimedia-Projekte. *Beschreibung von Eigenschaften einer Software durch Zahlen*. [Online; accessed 17-June-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. 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.