INHALTSVERZEICHNIS

Teil I

THESIS

1

EINLEITUNG

1.1 MOTIVATION

Es stellt sich immer die Frage, wie man Software qualität für leute greifbar machen kann, die nicht unbedingt Software-Entwickler sind. Außerdem wie kann auf einen blick als Software-Entwickler die Qualität des Codes einschätzen, ohne den Code gelesen zu haben bzw wie kann ich einen möglichst einfachen und schnellen einstieg in Code bekommen, um zu verstehen, wo ich anfangen muss zu verbessern. Es gibt unzählige Möglichkeiten, Software-Qualität zu visualisieren. Sei es in einfachen Diagrammen oder komplexen interaktiven Dashboards. Es gibt bereits viele ansätze code zu visualisieren und es gibt auch viele Ansätze dies drei dimensional zu tun, um die greifbarkeit und die plastizität zu erhöhen, um so das Verständnis zu fördern [MFM03; WL07; Rei95; KM00]. Der Vorteil von 3D Visualisierungen ist, dass sie eine wirklich räumliche Vorstellung des Codes Ermöglichen und eine beinahe immersive Erfahrung bieten. Dies kann helfen, Muster und Strukturen im Code zu erkennen, die in 2D-Darstellungen nicht abgebildet werden können. Außerdem lassen sich durch die Zusätzliche Dimensionen mehr Informationen in einem einzigen Bild darstellen. Weit verbreitet ist die Nutzung von 3D-STadt-Methaphern [WL07], um Software-Qualität zu visualisieren. Diese Methoden haben aber aktuell immernoch einige Probleme, speziell, wenn es um die Übersichtlichkeit geht [LF08]. Viele dieser Ansätze nutzen Treemaps-Layout und Algorithmen. Die Frage ist ob herkömmliche Treemaps-Algorithmen überhaupt geeignet sind, um wirklich übersichliche Treemap-Layouts zu erzeugen, oder ob nicht andere Ansätze besser geeignet sind. Dies wurde bisher noch nicht ausreichend untersucht.

Warum stadt? es ist einfach und mit 3d blöcken anders als zb. wald metaphern[Atz+21]

1.2 GRUNDLAGEN

1.2.1 *Software-Qualitätsmetriken*

Die ISO/IEC 25010 Norm [Iso] beschreibt wie Software-Attribute und -Qualitäten miteinander in Beziehung stehen. Diese Norm zerlegt die Software-Qualität in acht Qualitätsmerkmale, wie zum Beispiel funktionale Eignung. Dies ist der Blick von außen auf die Software, wir interessieren uns allerdings für den Blick auf die Software von innen, wenn wir also von Software-Qualität sprechen, dann sprechen wir von der Qualität des Codes und darüber wie man diese bewerten und verständlich machen kann.

Qualitätsmetriken sind Kennzahlen, die zur Bewertung der Qualität verwendet werden. Sie sind ein wichtiges Werkzeug zur Analyse und Verbesserung jeglicher Dinge. Im grunde sind metriken einfach nur messungen, die eine bestimmte Aussagekraft haben. OFt ist es nicht möglich eine Metrik zu identifizieren, die eine vollumfängliche aussage über etwas trifft. Oft ist es notwenig verschiedene metriken im zusammenspiel miteinander zu betrachten, um ein verständis für aspekte zu erhalten.

Dies gilt auch für Software-Qualitätsmetriken: "Eine Softwarequalitätsmetrik ist eine Funktion, die eine Software-Einheit in einen Zahlenwert abbildet, welcher als Erfüllungsgrad einer Qualitätseigenschaft der Software-Einheit interpretierbar ist."[EH98]

Es ist nicht möglich eine einzige Metrik zu finden, die eine vollumfängliche aussage über die Qualität einer Software trifft - was ist schon gute Software? Wann ist software gut? SEblst wenn man als Ziel von software definieren würde, dass sie fehlerfrei ist, wäre es nicht möglich eine einzige Metrik zu finden, die dies beschreibt. Wann ist eine software fehlerfrei? Ist sie fehlerfrei, wenn sie keine Bugs hat? Wenn sie keine Bugs hat, aber trotzdem nicht benutzbar ist, ist sie dann fehlerfrei?

Dieser Eigenschaft macht es notwendig verschiedene Metriken im zusammenspiel zu betrachten und sich einen übersblick über die Indizien zu verschaffen, die auf eine gute oder schlechte Qualität hinweisen. Die besonderheit bei Software-Qualitätsmetriken im vergleich zu anderen Metriken mag sein, dass die Daten hierarchisch strukturiert sind. Meist wird dies auf die orderner und datei-struktur abgebildet. So werden die Metriken in der Regel auf der Ebene der einzelnen Dateien ermittelt und dann auf die Ordner-Ebene aggregiert, wodurch eine hierarchische Baumstruktur entsteht.

Software-Qualitätsmetriken werden meist auf Software-Einheiten angewendet. Also z.B. auf Dateien des Quellcodes, auf Module oder auf Klassen. Diese Software-Einheiten sind in der Regel hierarchisch strukturiert. Auch die im folgenden vorgesllten ansätze zur Visualisierung von hierarchischen Daten sind nicht speziell für Software-Qualitätsmetriken gedacht, sondern stellen allgemeine Ansätze dar. Ich behaupte, dass die Sturktur von Software gemeinsamkeiten hat, zB. ein ordner hat nicht mehr als 20 Kinder, was interessante annahmen darstellt, die eventuell in die visualierungs algorithmen verbessernd einfließen können.

1.2.2 *Software-Visualisierung*

1.2.3 *3D-Software Visualisierung*

Warum 3D? Ist es nicht ausreichend software-qualität einfach als zahlenwerte darzustellen oder zumindest in 2D darzustellen? wie in Abschnitt 1.4 aufgezeigt, wollen wir software qualität auch für nicht-software-entwickler greifbar machen, einfache werte sind dafür nicht ausreichend.

> 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 öutside". 3D visualizations on the other hand provide the potential to create such an immersive experience... [WL07, S. 1]

2D visualisierung bietet zwar bestimmt einen guten überblick, aber macht die software nicht wirklich greifbar.

Die Idee Software in 3D darzustellen ist nicht neu. Schon 1995 stellte Steven P. Reiss einen Ansatz vor, der es ermöglichte Software in 3D darzustellen [Rei95]. Die meisten dieser ersten Ansätze verfolgten das Ziel Software für Entwickler greifbar zu machen, die Struktur aufzuzeigen und einen Highlevel überblick über eine Software zu geben, was speziell Hilfreich ist, wenn Entwickler ein System neu kennenlernen müssen [YM98]. Diese Ansätze sind in der Regel nicht für die Visualisierung von Software-Qualitätsmetriken gedacht, sondern zielen darauf ab, die Struktur und den Aufbau der Software zu verdeutlichen (siehe Abbildung 1.1).

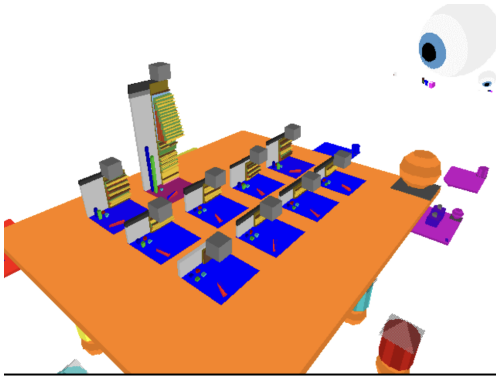

Abbildung 1.1: Beispiel für eine 3D-Visualisierung von Young und Munro [YM98, S. 6]

Trotzdem können wir auch für unser Ziel der Software-Qualitätsmetrik-Visualisierung Kriterien ableiten, die für eine gute Visualisierung wichtig sind [YM98]:

- **Darstellung**: Der wichtigste Aspekt ist die Darstellung der Software. Die Visualisierung sollte die Struktur und den Aufbau der Software verdeutlichen. Die Frage ist also, wie wird die Software dargestellt?
  - Informationsgehalt: Die Visualisierung sollte so viele Informationen wie möglich enthalten.

  - Niedrige visuelle Komplexität: Als Gegenspieler zum Informationsgehalt steht die visuelle Komplexität. Die Visualisierung sollte so einfach wie möglich gehalten werden, um den Betrachter nicht zu überfordern.
  - Skalierbarkeit: Die Visualisierung sollte auch bei großen Software-Systemen noch gut lesbar sein. Dies ist besonders wichtig, da wir hier über große Software-Systeme sprechen. Die Autoren von *Visualising Software in virtual reality* [YM98] sagen zudem, dass mechanismen Nötig sind, um Komplexität und Informationsgehalt zu steuern und je nach Software-System anpassen zu können.
  - Stabilität gegenüber Änderungen: Die Visualisierung sollte stabil gegenüber Änderungen in der Software sein. Das bedeutet, dass die Visualisierung sich nur so sehr wie nötig ändert, wenn sich die Software ändert, um eine Versions konsistente Vergleichbarkeit zu ermöglichen und bereits mit der Visualierung vertraute Nutzer nicht zu überfordern.
  - Gute Visuelle Metaphern: Die Visualisierung sollte gute visuelle Metaphern verwenden, um bereits bekannte Konzepte zu verwenden, um die Software verständlicher zu machen.

- **Abstraktion**: Das Ziel von Visualierung muss sein, unwichtige Details auszublenden und ein verständliches Modell der Software zu erstellen.
- **Navigation**: Da wir häufig über große Software-Systeme sprechen, kann es schnall passeiren dass Nutzer in der Visualisierung verloren gehen. Es muss also möglich sein, sich gut zurecht zu finden und intuitiv zu wissen, wo was ist, um so ein grfühl für die software zu erhalten.
- **Korrelation mit dem Code**: Die Visualisierung sollte eine gute Korrelation mit dem Code haben. Wenn man die Visualierung sieht, soll man diese auch mit dem Code in Verbindung bringen können. Es sollte also möglich sein, die Visualisierung mit dem Code zu verknüpfen und so ein besseres Verständnis für die Software zu bekommen.
- **Automatisierung**: Die Visualisierung sollte automatisiert werden können - ein Punkt der trivialier weise gegeben ist, da wir hier über algorithmen sprechen, die keine manuelle eingabe benötigen.

1.2.3.1 *CodeCity*

Der Cornerstone von 3D Software-Qualitätsvisualisierung ist das Konzept von CodeCity [WL07]. Das Paper von Richard Wettel und Michele Lanza verbindet einige Aspekte, die zu diesem Zeitpunkt neu waren. Sie reden zwar nicht konkret von software-qualitäts-visualierung, nutzen aber trotzdem qualitätsmetriken auf Klassenebene, um die richtige Granulariät von Software visualierung zu finden. Herkommliche paper waren oft noch auf niedrigeren ebenen z.b. Marcus Adrian et al. auf Ausdrucks-Ebene [MFM03]. Zudem nutzen sie, wie von Young und Munro [YM98] gefordert, eine gute

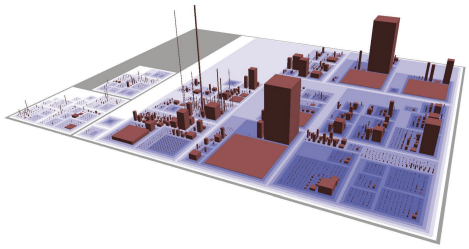

Abbildung 1.2: Beispiel für eine original CodeCity-Visualisierung [WL07, S. 2]

visuelle Metapher, um die Software darzustellen. Sie nutzen das Konzept einer Stadt, um die Software darzustellen. Dabei wird jede Klasse als Gebäude dargestellt, dabei hat jedes Artefakt (Klassen, Pakete und Ordner) verschiedene Attribute: Die Dimension, die Position, die Farbe, die Farbsättigung und die Transparenz. Ein Beispiel für eine solche Visualisierung ist in Abbildung 1.2 dargestellt.

Das Layout der Stadt wird lässt sich dabei herunterbrechen auf ein 2D-Layout Problem. Um dieses problem zu lösen, implementieren sie eine abwandlung von Treemap-Algoithmen (Was treemaps sind wird in abschnit 1.2.4 erklärt). Auffällig bei Ihrer implementierung ist, dass es viele ungenutzte leere flächen ohne gebäude gibt und gebäude anhand ihrer Größe sortiert plaziert werden (groß weiter unten links, klein weiter oben rechts). Leider ist der algotihmus nicht frei zugänglich, sodass in dieser Arbeit zum vergleich eine eigene Implementierung des Algorithmus verwendet wird, wodurch natürlich auch die Ergebnisse, von der original implementierung abweichen werden (außer wenn anders gekennzeichnet).

1.2.4 *Treemap-Layouts*

Eine Treemap visualisiert einen Baum, indem jedem Knoten ein Rechteck mit der Fläche A zugewiesen wird, proportional zu seinem zugewiesenen Wert (z.B. Datenmenge oder Marktwert). Nicht-Blatt-Knoten werden dabei üblicherweise durch Rahmen (Container-Rectangles) gekennzeichnet, um die Gruppierung der Kinder zu zeigen. [BHVW00] Die Rechtecke aller Blätter füllen die Fläche des Wurzelrechtecks vollständig aus. Mathematisch entspricht die Eingabedatenstruktur einem gewichteten Baum, bei dem jede Blatteinheit eine numerische Größe hat. Die Fläche eines Eltern-Rechtecks entspricht der Summe der Flächen (Werte) seiner Kinder.

Die konkrete Idee hierarchische DAten in form von Treemaps darzustellen wurde erstmals 1991 von Shneiderman und Johnson [JS91] vorgestellt. Sie stellten fest, dass die Darstellung von hierarchischen Daten in Form von Bäumen in der Regel nicht sehr anschaulich ist. Sie entwickelten eine Methode, um diese Daten in Form von Rechtecken darzustellen, die die Fläche der Knoten proportional zu ihrem Wert darstellen. Diese Methode wurde als "Treemap" bezeichnet. Als Ziele dieser Visualierung formulierten sie unter anderem diese Aspekte:

- **Effiziente Nutzung des Platzes:** Generell soll es darum gehen möglichst viele Informationen auf einem kleinen Raum darzustellen.
- **Verständlichkeit:** Die Visualisierung soll so gestaltet sein, dass sie für den Betrachter leicht verständlich ist. Es soll möglich sein schnell und mit nur niedrigem kognitiven Aufwand die dargesllten Informationen zu erfassen.
- **Ästethik:** Die Visualisierung soll ansprechend gestaltet sein.

Zuvor bestehende Ansätze zur Visualisierung von hierarchischen Daten waren in der Regel nicht sehr anschaulich, besonders, wenn es um große Datenmengen ging. Listen, Baumdiagramme (siehe Abbildung 1.3) oder andere Darstellungen (auch bekannt als Node oder Link-Diagramme) sind nicht in der lage alle diese Aspekte zu erfüllen. Bei Einem typischen Baumdiagram zum Beispiel werden teilweise mehr als die Hälfte der Fläche für Hintergrund genutzt [JS91, S. 3] außerdem ist es schwer, außer der Struktur der Daten auch die Metriken darzustellen. Sie kritisieren auch die Darstellung von hierarchischen Daten in Form von Venn-Diagrammen (siehe Abbildung 1.4): "The space required between regions would certainly preclude this Venn diagram representation from serious consideration for larger structures."[JS91, S. 5] Es ist zwar möglich durch die Größe der Kreise eine Metrik darzustellen, es sei aber nicht möglich eine große Anzahl an Knoten sinnvoll darzustellen.

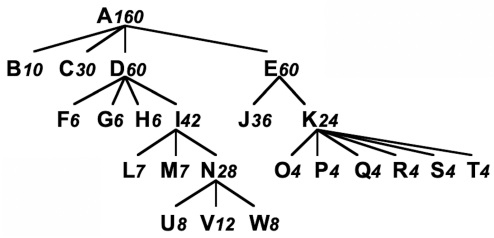

Abbildung 1.3: Beispiel für ein Baumdiagramm

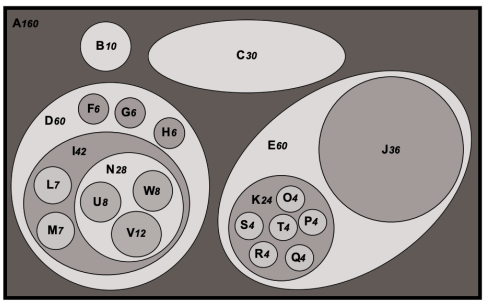

Abbildung 1.4: Beispiel für ein Venn-Diagramm

"Using boxes instead of ovals and a bin-packing algorithm could partially solve this space problem. But bin-packing is an NP-complete problem and does not preserve order."[JS91, S. 5] Sie stellen fest, dass es theoretisch eine dem Venn Diagramm ähnliche Lösung gibt, die allerdings NP-Hard ist.

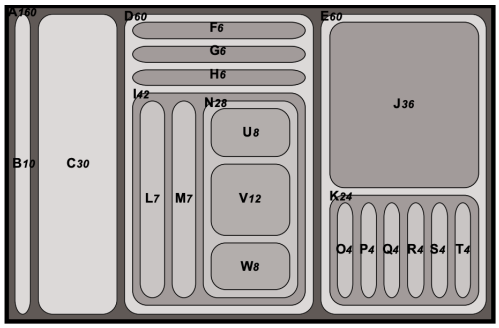

Abbildung 1.5: Beispiel für ein Boxed Venn-Diagramm

Shneiderman und Johnson schlagen zur Lösung dieser Schwierigkeiten ihren Treemap ansatz vor. Sie legen vier Eigenschaften fest, die bei der Erstellung der Treemaps gewährleistet werden:

- Wenn ein Knoten 1 ein Vorfahre von Knoten 2 ist, dann ist der Bereich von Knoten 1 vollständig enthalten in dem Bereich von Knoten 2.
- Die Bereiche von zwei Knoten schneiden sich, wenn ein Knoten ein Vorfahre des anderen ist.
- Knoten belegen eine Fläche, die streng proportional zu ihrem Gewicht ist.
- Das Gewicht eines Knotens ist größer oder gleich der Summe der Gewichte seiner Kinder.

Sie stellen auch einen Algorithmus vor, der diese Eigenschaften erfüllen soll. Ein von diesem Algorithmus erzeugtes Layout ist in Abbildung 1.6 dargestellt.

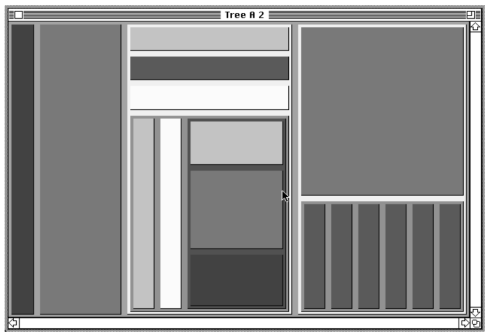

Abbildung 1.6: Beispiel für eine Treemap

Der Algorithmus unterteilt den Raum abwechselnd vertikal und horizontal, je nach größe der Knoten. Der Algorithmus arbeitet sich rekursiv von der Wurzel bis zu den Blättern herunter und hat eine Laufzeit von O(n), wobei n die Anzahl der Knoten ist. Im nächsten Abschnitt wird ein ähnlicher Algorithmus im Detail beschrieben, was zum besseren verständis dieser Art von Algorithmen helfen wird.

Obwohl es sich hier um ein renomiertes und viel Zitiertes Paper handelt, machen die Authoren einen entscheidenden Fehler, der in manchen Fällen sogar dazu führen kann, dass Rechtecke komplett verschwinden. Die Authoren betrachten dieses Problem in ihrem Paper leider nicht. Der Abstand zwischen den Knoten wird nämlich dadurch erzeugt, dass die Rechtecke diesen Abstand links, recht, oben und unten abgezogen bekommen. Dadurch ist die dargestellte Fläche der Rechtecke nicht mehr proportional zu den Werten, die sie darstellen sollen. Eigenschaft 3 wird also verletzt. Dies ist besonders problematisch, wenn die Rechtecke sehr klein bzw. sehr langgezogen sind. In diesem Fall kann es dazu kommen, dass die Rechtecke so klein werden, dass sie nicht mehr dargestellt werden können. Dies stellt in der Praxis ein riesiges Problem dar, speziell wenn das Problem dieser Arbeit also 3D im kopf bahalten wird. Es kann ja theoretisch vorkommen, dass als Flächenmetrik die lines of code verwendet wird und dort ein file, weil er wenige lines hat sehr klein wird und deswegen aufgrund der margins nicht angezeigt wird. Wenn jetzt aber die metrik für die Höhen berechnung die prozentualle test abdeckung der lines ist und der file nicht getestet ist, dann würde ein potentiell großes problem, was auch eigentlich direkt in auge springen sollte, nicht angezeigt werden.

Ein weiteres Problem bei diesem Algorithmus ist auch generell, dass unter umständen die Rechtecke sehr langgezogen werden können, was unter Umständen auch Kriterium 3 der Ästethik verletzt. Es gibt viele Ideen dieses Problem anzugehen. Eine Möglichkeit, den Squarify Algorithmus zu verwenden, wird im nöchsten Abschnitt vorgestellt.

1.2.4.1 *Squarify-Algorithmus*

Der Squarify-Algorithmus ist ein Layout-Algorithmus für Treemaps, der darauf abzielt, die Fläche der Rechtecke so ausgewogen wie möglich zu gestalten. Beduetet, dass die Rechtecke möglichst quadratisch sind. Die ursprüngliche form des algorithmusses wurde im Jahr 2000 von Bruls et al. [BHVW00] vorgestellt. Sie stellten fest: “another problem of standard treemaps [is] the emergence of thin, elongated rectangles”[BHVW00, S. 1]. wenn rechtecke nicht mehr so langgezogen sind, Es ist einfacher auf Rechtecke zu zeigen, diese wahrzunehmen, sie zu vergleichen und ihre größe einzuschätzen.

Der Algorithmus arbeitet rekursiv und teilt die Fläche in Rechtecke auf, wobei er versucht, die Seitenverhältnisse der Rechtecke so nah wie möglich an 1 zu halten. Sie stellen einen rekursiven Ansatz vor, bei dem, wie auch bei den meisten anderen Treemap-Algorithmen, die Rechtecke von oben nach unten (also vom Wurzelknoten bis zu den Blattknoten) aufgeteilt werden.

Im Folgenden wird der Algorithmus beschrieben, da er eine wichtige Grundlage für das Verständnis des Problems darstellt und außerdem eine gute Grundlage zum Verständnis der anderen Algorithmen bietet, da viele Algorithmen ähnliche Ideen verwenden.

Der Algorithmus wird anhand eines Beispiels aus dem originalen Squarify-Paper [BHVW00, S. 5] erläutert. Wir werden den Algorithmus jedoch anders erklären als im Paper, da wir uns näher an der Implementierung orientieren,

wie sie in der bekannten d3-Bibliothek [D3t] umgesetzt ist. Es sollen Rechtecke mit den Größen 6, 6, 4, 3, 2, 2, 1 in ein 6 mal 4 Rechteck einsortiert werden.

Der algorithmus arbeitet immer in Reihen, die er versucht zu füllen und dabei die Rechtecke möglichst quadratisch zu halten. Das erste Rechteck ist breiter als lang. (In dieser Arbeit werden wir, anders als in den herkömmlichen papern zu layout algorithmen, das Wort breit als x koordinate und das wort lang als y koordinate nutzen - das hat den hintergrund, dass das wort hoch im drei dimensionalen meist für die z komponent genutzt wird und es sonst zu verwirrungen kommen könnte) Da das Rechteck in das wir einfügen breiter als lang ist, werden wir eine imaginäre horizontale Reihe so lange mit Rechtecken befüllen, bis ein Threshold erreicht ist. Das erste Rechteck mit größe 6 fügen wir also in Schritt 1 in diese Reihe ein. Das Seitenverhältnis dieses Rechtecks beträgt 8 zu 3 (das Rechteck ist 1.5 Einheiten breit und 4 Einheiten lang). Das zweite Rechteck mit größe 6 fügen wir in Schritt 2 in diese horizontale Reihe ein, dabei wird die Reihe entsprechend breiter. Das Seitenverhältnis des jetzt eingefügten Rechecks beträgt 3 zu 2 (das Rechteck ist 3 Einheiten breit und 2 Einheiten lang). Jetzt kommt das nächste Rechteck mit größe 4. Dieses Rechteck ist hat ein Seitenverhältnis von 4 zu 1 (das Rechteck ist 4 Einheiten breit und 1 Einheit lang). Das hinzufügen dieses Rechtecks führt nun aber dazu, dass das schlechteste Seitenverhältnis der Reihe von 3 zu 2 auf 4 zu 1 ansteigt. Deshalb wird diese Reihe als abgeschlossen angesehen und die nächste Reihe wird begonnen - Schritt 4.

Dieser Schritt des suchens des schlechtesten Seitenverhältnisses lässt sich von der rechenkomplexität her gut optimieren, sodass der Ratio berechnet werden kann, ohne dass die Reihe wirklich mit Rechtecken gefüllt werden muss. Anstatt für jeder Rechteck max(w/l, l/w) zu berechnen, ziehen wir folgende vereinfachung heran.

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

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

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

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

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

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

Analog dazu gilt das gleich auch für $\frac{l_i}{w_i} = \frac{sV^2}{V_i \cdot w^2}$. Da wir nur an dem maximalen Wert beider Ausdrücke interessiert sind und die Länge (l_i) aller Rechtecke in der Reihe gleich ist, reicht es den Wert für das größte und das kleinste Rechteck zu berechnen und davon den maximalen Wert zu nehmen.

w ist für den gesamten Zeitraum des füllens einer Reihe konstant und muss daher nur einmal berechnet werden. sV wird mit jedem Rechteck aktualisiert.

Das Ziel ist es das Verhältnis der Seitenlängen gleich zu halten. Im ursprünglichen Paper von Bruls et al. [BHVWoo] wird darauf noch nicht so eingegangen, aber viele implementierungen z.b. die von d3.js [D3t] ermöglichen es, das Verhältnis nicht nur an den Wert 1 anzunähern, sondern auch an andere Werte, zum Beispiel den goldenen Schnitt.

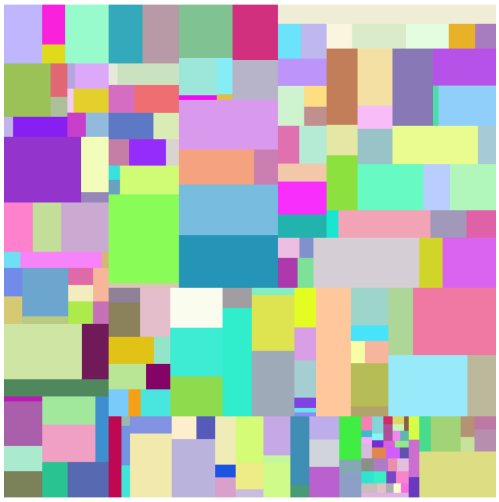

Abbildung 1.7: Beispiel für ein Squarify-Layout mit Annäherung an quadratische Rechtecke (Durchschnittliches Seitenverhältnis 1.42)

Abbildung 1.8: Beispiel für ein Squarify-Layout mit Annäherung an den Wert 5 (Durchschnittliches Seitenverhältnis 2.79)

1.3 PROBLEMSTELLUNG

Die fundamentale und schwerste Frage, bei den Stadt-Analogien ist, wie das Layout der Stadt aussehen soll.

1.3.1 *Das Treemap Problem*

In Abschnnit 1.2.4 wurde aufgezeigt, dass bereits der initiale Algorithmus von Johnson und Shneiderman [JS91] ein fundamentales Problem aufweist, wenn Treemaps mit Abständen zwischen Knoten dargestellt werden sollen.

- Da der Abstand von der Fläche der Knoten abgezogen wird, ist die dargestellte Fläche nicht mehr proportional zum Wert des Knotens.
- Durch das Abziehen der Abstände kann es passieren, dass Knoten verschwinden, wenn entweder die Länge oder die Breite der Knoten kleiner oder gleich dem Abstand ist.

In den Abbildungen 1.9, 1.10 und 1.11 sind Treemap Layouts abgebildet, die mit dem Squarify Algorithmus nach Abschnitt 1.2.4.1 generiert wurden. Der Visualierten Metriken sind händisch erstellt, um das Problem zu verdeutlichen (siehe Anhang HIER REF EINRÜGEN). In Bild 1.9 sind alle Knoten sichtbar und die Fläche der Knoten ist proportional zu den Werten der Knoten. In Bild 1.10 sind immernoch alle Knoten sichtbar, aber die Fläche der Knoten ist nicht mehr proportional zu den Werten der Knoten. Zum Beispiel hat der Große Knoten mit Wert 3000 eine Fläche von ca. 2600, wärend der kleine Knoten oben links mit Wert 30 nur eine Fläche von ca. 5 hat. In Bild 1.11 mit Abstand 10 sind dann schon einige Knoten (zum beispiel der Eben genannte knoten oben links) nicht mehr sichtbar, da die breite der Knoten kleiner als der Abstand ist.

Es ist nicht trivial dieses Problem zu lösen, da es die grundlegende Annahme der Treemap Algorithmen verletzt, dass die Fläche aller Knoten bekannt ist, bevor die Knoten plaziert werden. Bevor die Knoten plaziert werden, ist nicht klar, wie die Fläche der Knoten aussieht, das heißt, es ist auch nicht klar, wie viel Platz für die Abstände zwischen den Knoten benötigt wird. Dies wird klar wenn man sich die Abbildung 1.12 anschaut. Dort sieht man, dass die Fläche, die für Knoten in ihren Eltern benötigt wird, größer ist als die Fläche, die für die Knoten selbst benötigt wird und diese benötigte Fläche stark vom Layout der Knoten selbst abhängt. Somit ist auch unklar, wie groß die Fläche aller elternknoten sind.

Das Problem ist jetzt aber, dass wenn die Fläche der Knoten nicht bekannt ist auch das Layout der Knoten nicht berechnet werden kann, da die Fläche der Knoten für das Layout benötigt wird. Hier ergibt sich also ein Zirkelschluss: Die Fläche ist nicht klar, ohne das Layout und das Layout kann nicht berechnet werden, ohne die Fläche zu kennen.

Abbildung 1.9: Treemap Layout generiert mit dem Squarify Algorithmus nach Abschnitt 1.2.4.1 mit einem Abstand von 0

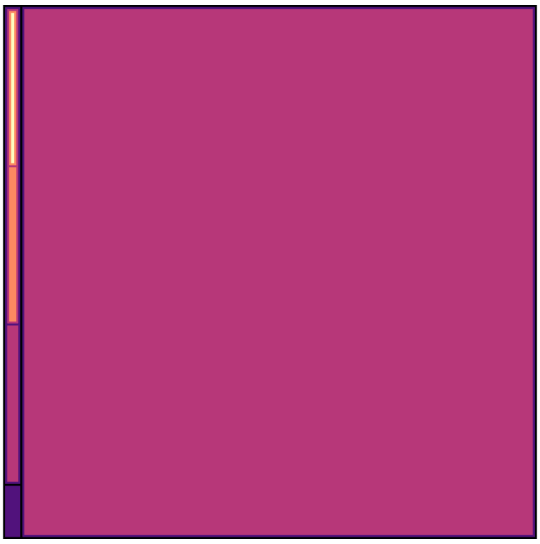

Abbildung 1.10: Treemap Layout generiert mit dem Squarify Algorithmus nach Abschnitt 1.2.4.1 mit einem Abstand von 5

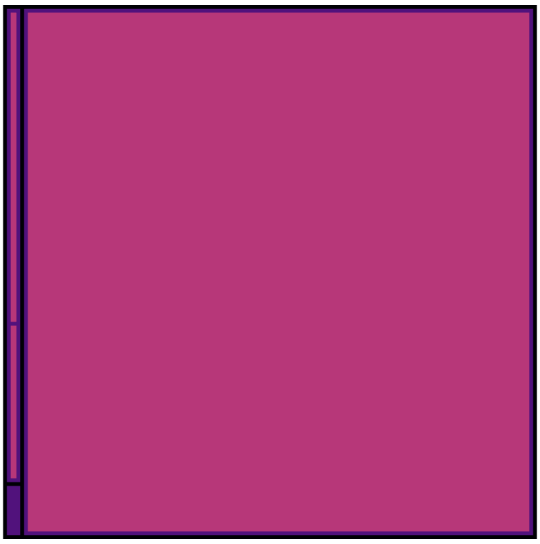

Abbildung 1.11: Treemap Layout generiert mit dem Squarify Algorithmus nach Abschnitt 1.2.4.1 mit einem Abstand von 10

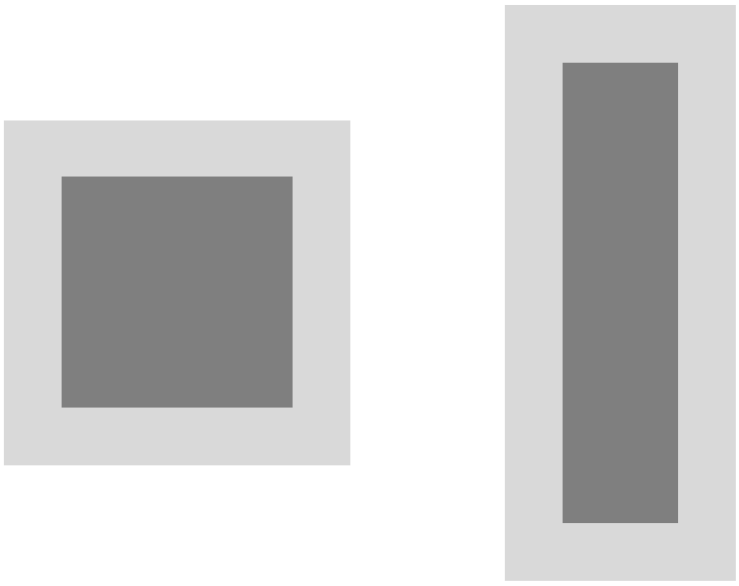

Abbildung 1.12: Abbildung eines zweier Rechecke mit der Fläche 16 in dunkel grau und in hellgrau der Abstand von 1 um die Flächen herum. Das Linke Rechteck (4x4) mit Abstand nimmt eine Fläche von 25 (5x5) ein. Das rechte Rechteck (2x8) mit Abstand nimmt eine Fläche von 40 (4x10) ein.

1.4 ZIEL

Laut Marcus Adrian et al. gibt es 5 Dimensionen die man beachten muss, wenn es um software visualierung geht:

> • Tasks - why is the visualization needed? • Audience - who will use the visualization? • Target - what is the data source to represent? • Representation - how to represent it? • Medium - where to represent the visualization? [MFM03, S. 2]

Wir beantworten diese Fragen, um das Ziel dieser Arbeit zu begründen. **Warum ist die Visualisierung von Codequalitätsmetriken wichtig?** Die Visualisierung von Codequalitätsmetriken ist wichtig, um die Qualität von Softwareprojekten zu bewerten und zu verstehen, aber auch um einen schnellen Überblick über die Codebasis zu geben und einen Einstieg in vertiefende Codeanalysen zu ermöglichen. Eine effektive Visualisierung kann helfen, schnell Hotspots im Code zu identifizieren, die möglicherweise verbessert werden müssen, und somit die Wartbarkeit und Qualität des Codes zu erhöhen. Außerdem soll die Visualisierung ermöglichen verschiedene Metriken in verbindung zu setzen, um so eine höhere Aussagekraft über den Code, den die einzelne betrachtung jeder Metrik nicht bieten kann. Der wichtigste Punkt ist, das subjektive *Greifbar* machend der CodeQualität.

Wer wird die Visualisierung nutzen? Die Visualisierung ist vorallem an Personen gerichtet, die sich nicht mit der Codebasis auskennen. Das können Entscheidungsträger sein, die keine ahung von software entwicklung haben, das können aber auch entwickler sein, die sich neu in ein Projekt einarbeiten müssen, um die qualität einer software zu erhöhen.

Was ist die Datenquelle? Die Datenquelle sind hierarschiche Codequalitätsmetriken, die aus dem Quellcode eines Softwareprojekts extrahiert werden. Dabei wird jeder Knoten in dieser Hierarchie als "Node"bezeichnet. Jede Node hat folgendes Schema: “‘json "node": "name": string, "children": List[Node] | "value": number, “‘

Wo soll die Visualisierung dargestellt werden? Die Visualisierung soll digital auf herkömmlichen Bildschirmen dargestellt werden. Speziell wird in dieser Arbeit beispielhaft eine darstellung in einem Webbrowser angestrebt und die Algorithmen in Typescript implementiert. Natürlich können aber alle Ergebnisse auch in anderen Programmiersprachen und Umgebungen umgesetzt werden.

Wie soll die Visualisierung dargestellt werden? Im GRunde soll eine Visualierung in Anlehnung an den in Abschnitt 1.2.3.1 beschriebenen Stadt-Metapher ansatz verfolgt werden. - Speziell soll es in dieser Arbeit um das Layout der Knoten gehen, aber im hinterkopf soll die stadtmetapher bleiben und immer als grundlage für die bewrtung des 2d layouts dienen. Wie in der CodeCity arbeit beschrieben, soll es möglich sein Metriken in form von Fläche, Höhe und farbe (ob jetzt nur farbe oder durchsichtigkeit wie bei dem codecity paper oder sogar textur von knoten - wird hier ignoriert). Das heißt also, dass das 2D layout in gewisser weise eingeschtränkt wird, zB.

wenn Farbe als visulaisrung von struktur verwendet werden soll oder wie in [BHVWoo] schattierung.

Speziell soll die Visualisierung am Ende optimiert auf folgende Aspekte sein, die sich aus den im Abschnitt 1.2 beschriebenen Grundlegenden Aspekten von Softwarevisualierung, leicht angepasst and das Problem dieser Arbeit, ableiten lassen:

- **Informationsgehalt und Effiziente Nutzung des Platzes:** Die Visualisierung sollte so viele Informationen wie möglich auf so wenig Platz wie möglich darstellen.
- **Niedrige visuelle Komplexität und Verständlichkeit:** Als Gegenspieler zum Informationsgehalt steht die visuelle Komplexität. Die Visualisierung sollte so einfach und verständlich wie möglich gehalten werden, um den Betrachter nicht zu überfordern.
- **Skalierbarkeit:** Die Visualisierung sollte auch bei großen Software-Systemen noch gut lesbar sein. Dies ist besonders wichtig, da wir hier über große Software-Systeme sprechen.
- **Korrelation mit dem Code:** Die Visualisierung sollte eine gute Korrelation mit dem Code haben. Wenn man die Visualierung sieht, soll man diese auch mit dem Code in Verbindung bringen können. Es sollte also möglich sein, die Visualisierung mit dem Code zu verknüpfen und so ein besseres Verständnis für die Software zu bekommen.
- **Zweitrangig ist Stabilität gegenüber Änderungen:** Die Visualisierung sollte stabil gegenüber Änderungen in der Software sein, damit der Qualitätszustand der Software einfacher über die Zeit verfolgt werden kann.

2

HAUPTTEIL

2.1 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 1.3.1 beschrieben wurde, anzugehen.

2.1.1 *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.

2.1.2 *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.

2.1.2.1 *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.

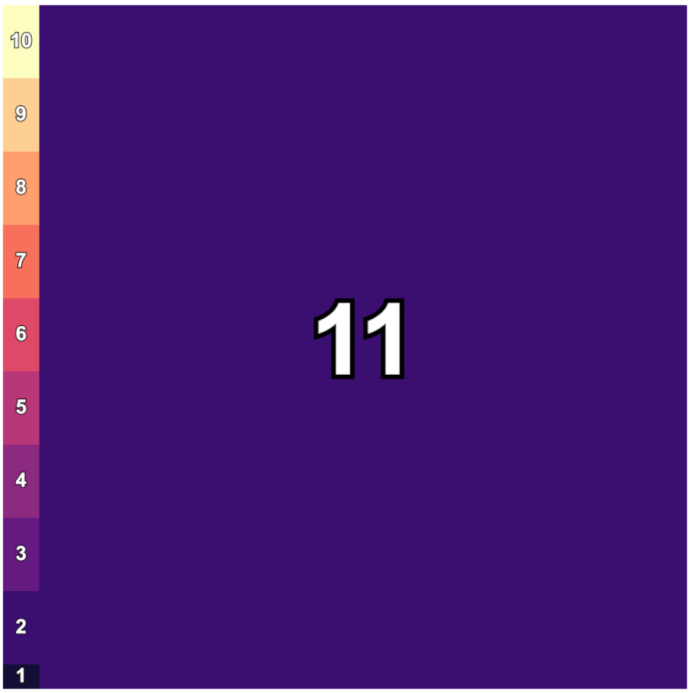

Abbildung 2.1: Treemap Layout generiert mit dem Squarify Algorithmus nach Abschnitt 1.2.4.1 mit einem Abstand von 0 und der einfachen Größenanpassung auf der händisch erstellen map (siehe Anhang).

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
```

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 angenom-

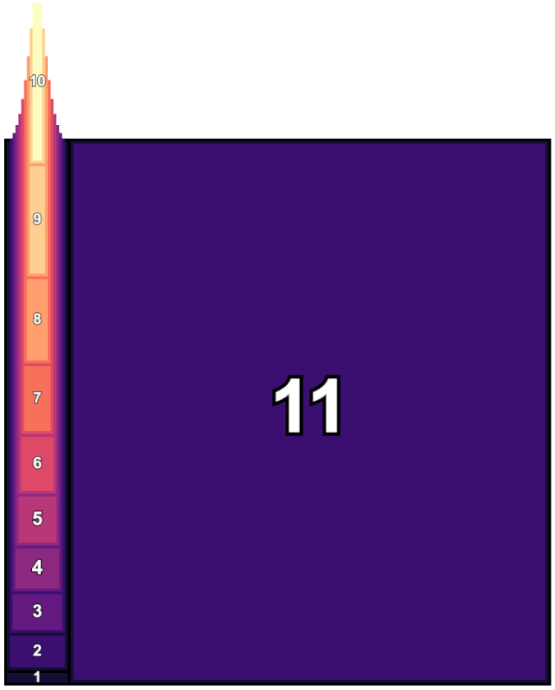

Abbildung 2.2: Treemap Layout generiert mit dem Squarify Algorithmus nach Abschnitt 1.2.4.1 mit einem Abstand von 1 und der einfachen Größenanpassung auf der händisch erstellen map (siehe Anhang).

men, 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.

Dieser Effekt kann einfach behoben werden, wie wir in Abschnitt 2.1.3 sehen werden.

2.1.2.2 *Relative Größenanpassung*

Der Algorthmus wird im foldenden als Pseudocode dargestellt (siehe Algorithmus 2).

Algorithm 2 Größenanpassung

```
1:  function increaseValues(node: SquarifyNode, margin: number)
2:      if node.value = 0 then
3:          return 0
4:      end if
5:      childrenValueIncrease ← 0
6:      if node.children then
7:          for child in node.children do
8:              childrenValueIncrease += increaseValues(child, margin)
9:          end for
10:     end if
11:     oldValue ← node.value
12:     ratioChildrenValueIncrease ← (oldValue + childrenValueIncrease) /
    oldValue
13:     newWidth ← sqrt(ratioChildrenValueIncrease) * currentWidth
14:     newLength ← sqrt(ratioChildrenValueIncrease) * currentLength
15:     newWidth += margin * 2
16:     newLength += margin * 2
17:     node.value ← newWidth * newLength
18:     return node.value - oldValue
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 2).

2.1.3 *Scaling der Knoten*

Wenn die Fläche innnerhalb von Elternknoten immer kleiner wird, kann es passieren, dass Knoten über ihre Elternknoten hinauswachsen, wie in Abbildung 2.2 zu sehen ist. Es kann genauso passieren, dass die Fläche innerhalb der Knoten größer wird, wie in Abbildung HIER BEISPIEL ZEIGEN 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 2.3 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. Persönlich würde ich sagen, dass dieser Effekt für

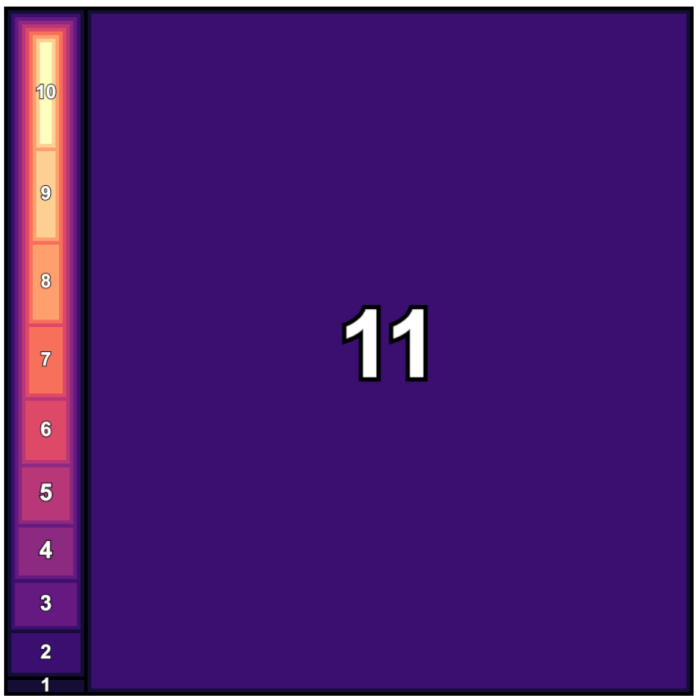

Abbildung 2.3: Treemap Layout generiert mit dem Squarify Algorithmus nach Abschnitt 1.2.4.1 mit einem Abstand von 1 und der einfachen Größenanpassung auf der händisch erstellen map (siehe Anhang) und der Skalierung der Knoten.

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.

2.1.4 *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.

2.1.5 *Fazit*

Immernoch straight forward, es gibt aber noch Probleme

3

SCHLUSS

3.1 EVALUATION

3.2 KRITERIEN FÜR CODECITY 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.
- **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.

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.

Teil II

APPENDIX

LITERATUR

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

[D3t] [Online; accessed 19-May-2025]. 2025. URL: https://d3js.org/d3-hierarchy/treemap.

[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.

[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.

[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.

[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.

[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.

[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.

[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.