Teil I

THESIS

0.1 GRUNDLAGEN

0.1.1 *Software-Qualitätsmetriken*

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.

0.1.2 *Software-Visualisierung*

• 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]

0.1.3 *3D-Software Visualisierung*

Warum 3D? Ist es nicht ausreichend software-qualität einfach als zahlenwerte darzustellen oder zumindest in 2D darzustellen? wie in Abschnitt **??** 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 **??**).

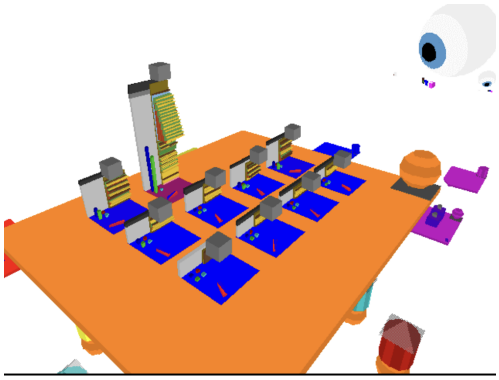

Abbildung 1: Beispiel für eine 3D-Visualisierung von Young und Munro [YM98, S. 6]

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.

0.1.4 *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 **??**) 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 **??**): "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.

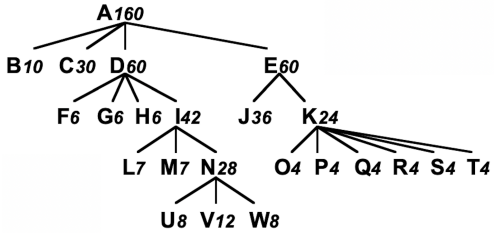

Abbildung 2: Beispiel für ein Baumdiagramm

"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.

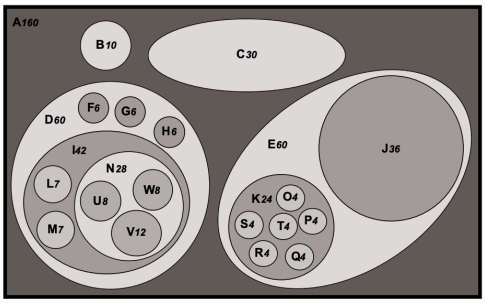

Abbildung 3: Beispiel für ein Venn-Diagramm

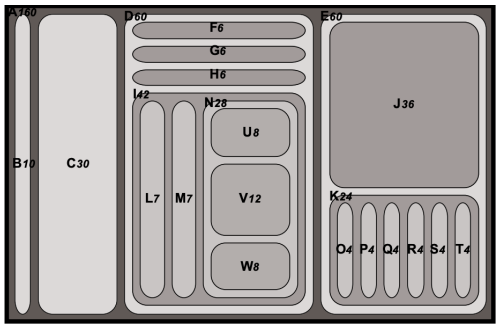

Abbildung 4: Beispiel für ein Boxed Venn-Diagramm

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 **??** dargestellt.

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

Abbildung 5: Beispiel für eine Treemap

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.

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.

$$\frac{w_i}{l_i} = \frac{w_i \cdot l_i \cdot w^2}{l_i \cdot l_i \cdot w^2} \tag{0.1}$$

$$= \frac{w_i \cdot l_i \cdot w^2}{l_i^2 \cdot \left(\sum_{j=0}^{n} w_j\right)^2} \tag{0.2}$$

$$= \frac{w_i \cdot l_i \cdot w^2}{\left(l_i \cdot \sum_{j=0}^{n} w_j\right)^2} \tag{0.3}$$

$$= \frac{w_i \cdot l_i \cdot w^2}{\left(\sum_{j=0}^{n} l_i \cdot w_j\right)^2} \tag{0.4}$$

$$= \frac{w_i \cdot l_i \cdot w^2}{\left(\sum_{j=0}^{n} l_j \cdot w_j\right)^2} \quad \text{da } \forall i, j \in \{0, \ldots, n\}, l_i = l_j \tag{0.5}$$

$$= \frac{V_i \cdot w^2}{sV^2} \tag{0.6}$$

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.

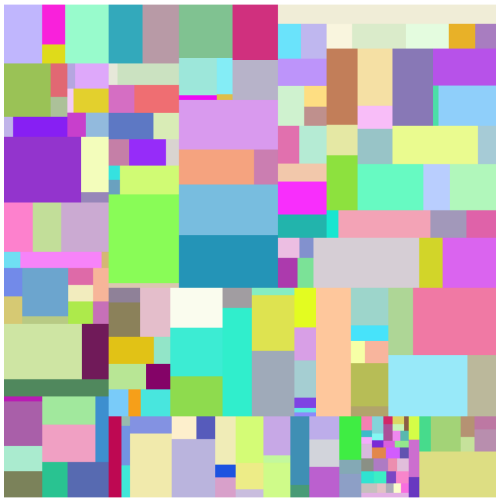

Abbildung 6: Beispiel für ein Squarify-Layout mit Annäherung an quadratische Rechtecke (Durchschnittliches Seitenverhältnis 1.42)

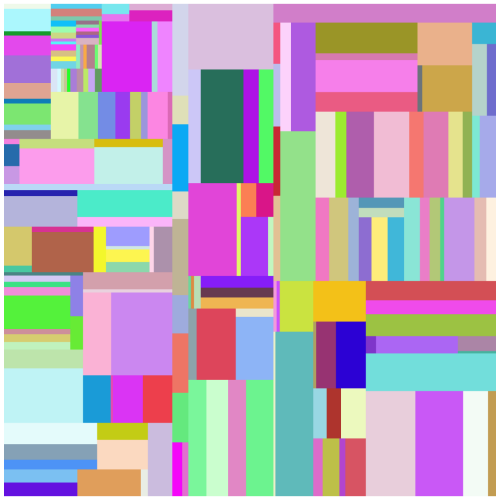

Abbildung 7: Beispiel für ein Squarify-Layout mit Annäherung an den Wert 5 (Durchschnittliches Seitenverhältnis 2.79)

Teil II

APPENDIX

LITERATUR

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

[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.

[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.