Visualisierung von Funktionsaufrufen in Software als räumliche Strukturen

Kai Jauslin, Gerhard M. Buurman, Jürgen Späth
Zürcher Hochschule der Künste, Departement Design
Interaction Design Programm
Ausstellungsstrasse 60, 8005 Zürich, Switzerland
{kai.jauslin,gerhard.buurman, juergen.spaeth}@zhdk.ch
Januar 2009

ÜBERSICHT

Ein Softwaresystem besteht aus modularen Teilen, die in vielfältiger Weise voneinander abhängig sind. Für einen ausgewählten funktionalen Aspekt der Software ist es schwierig, die Konsequenzen von lokalen Änderungen auf Performance und Stabilität der Gesamtarchitektur zu verstehen. In dieser Arbeit wird eine 3D–Visualisierung vorgestellt, welche ausgewählte Funktionsaufrufe einer Software in einer charakteristischen, räumlichen Struktur darstellt. Dadurch werden die Funktionsaufrufe untereinander vergleichbar. Über ihre visuellen Eigenschaften transportiert sie zusätzlich wertvolle Informationen zur Gesamtarchitektur der Software. Aus der Visualisierung können so durch interaktive Änderung des Blickwinkels Erkenntnisse zur Evolution und zum Verhalten des Funktionsaufrufs gewonnen werden.

1. EINFÜHRUNG

Die Entwicklung grosser Softwaresysteme ist eine schwierige Aufgabe. Moderne Programme bestehen aus Millionen von Codezeilen, verwoben in einer komplexen Architektur von Abhängigkeiten und Kommunikationswegen. Software ist per Definition immateriell und hat deshalb von sich aus keine bestimmte Form oder Materialeigenschaften. Das macht es einem Software–Entwickler schwer, Abläufe und Strukturen zu überblicken und die Gesamtzusammenhänge zu verstehen. Ein solches fehlendes Verständnis auf Seiten des Entwicklers kann zu fehleranfälliger und langsamer Software führen.

Eine Software definiert sich über ihren Quellcode. Dieser besteht aus einer statische Architektur in Form von Klassen, Modulen und Methoden. Komplementär zu diesem Code entstehen während der Ausführung des Softwareprogrammes dynamische Kommunikationspfade, welche keinen direkten Bezug zur Architektur haben müssen. Unter Umständen wird während der Ausführung einer Software nur ein kleiner Prozentsatz der Architektur überhaupt benutzt. Durch Messwerte basierend auf der Architektur und Grösse des Quellcodes alleine, lässt sich also nur ein Teil der Software verstehen.

Neben den Unterschieden im statischen und dynamischen Vernetzungsmuster, ist eine Software zusätzlich kontinuierlichen zeitlichen Veränderungsprozessen entworfen. Die sogenannte *Evolution* der Software zeigt sich in einem Versionsverlauf, einer Abfolge von Änderungen am Quellcode. Diese Änderungen sind meist zur einer Fehlerbehebung oder dem Hinzufügen neuer Funktionalitäten. Meist sind an einer solchen Änderung mehrere Entwickler beteiligt, was es schwer macht, den gegenseitigen Einfluss auf das Ausführungsverhalten im Voraus abschätzen zu können.

In unserer Arbeit entwickeln wir ein Konzept, welches die verschiedenen Dimensionen der Zeit, Architektur und Kommunikation als räumliche Strukturen abbilden kann. Wir gehen dabei von einer einzelnen Funktionalität aus, abgebildet durch einen Funktionsaufruf auf höherer Ebene. Dieser kann auch mittels einem Use–Case beschrieben werden.

Die entwickelte Struktur enthält charakteristische Eigenschaften sowohl der statischen Code–Architektur wie auch dem dynamischen Kommunikationsmuster und dem Versionsverlauf. Durch die interaktive Veränderung des Blickwinkels eines Betrachters auf die Struktur, können aufgrund der visuellen Eigenschaften unterschiedliche Erkenntnisse über den Funktionsaufruf gewonnen werden.

2. VERWANDTE ARBEITEN

Das Gebiet der Software–Visualisierung beschäftigt sich mit der automatisierbaren Darstellung von Software im zwei– und dreidimensionalen Raum. Die Visualisierungen bauen auf den folgenden Teilgebieten auf:

- Software–Architektur (Klassendiagramm)
- Verhalten zur Laufzeit (Performance, Kommunikation)
- Evolution (Versionsverlauf)

Die meisten Systeme bauen auf statischen Software–Metriken auf. Das sind Messwerte, welche aufgrund von automatisierten Analysen des Quellcodes berechnet werden können. Die Ausführung einer Software ist dazu nicht notwenig. Beispiele solcher Metriken sind die Anzahl der Zeilen im Quellcode (Lines of Code) oder die Cyclomatic Complexity – ein Mass für die Komplexität der Entscheidungspfade innerhalb einer Funktion.

In objektorientierten Systemen definiert die Metrik der *Instability* die globalen Abhängigkeiten einer Funktion mit einem Wert zwischen 0 und 1. Eine Funktion ist instabil, je mehr sie von anderen Funktionen abhängt. Eine absolut stabile Funktion ruft selbst keine anderen Funktionen auf. Der Wert der Instability berechnet sich aufgrund der Anzahl eingehender und ausgehender Funktionen, jeweils auf Basis der Gesamtarchitektur.

Diese statischen Metriken werden für jede *natürliche* Granularität der Software berechnet. Diese entsteht einerseits durch die Aufteilung des Quellcodes in Dateien und Verzeichnishierarchien, andererseits durch die Unterteilung in Funktionen, Klassen, Module und Pakete. Die Berechnung der Metriken kann direkt in einer Entwicklungsumgebung wie Microsoft Visual Studio [10] oder Eclipse mit Metrics [9] erfolgen.

Wichtige Projekte für die Visualisierung von statischen Software–Metriken sind:

- CodeCity [12]: die gesamte statische Architektur des Codes wird mit der Metapher einer Stadt im dreidimensionalen Raum abgebildet. Die Höhe und Farbe der Häuser variiert mit der Auswahl der Metriken.
- CocoViz [2]: aus der statischen Architektur einer Software wird eine räumliche Landschaft erzeugt. Jedes Modul wird als Form dargestellt deren gestalterische Ausprägung von mehreren Metriken beeinflusst wird. Unsere Wahrnehmung kann dann aufgrund von Erfahrungen eine Interpretation der Metrik vornehmen.

Weitere Arbeiten beschäftigen sich mit der dynamischen Visualisierung von Software. Dabei wird die Analyse während der Ausführung, also zur Laufzeit der Software, ausgeführt. Die Analyse startet mir einer bestimmten Funktion und speichern ein Protokoll der aufgerufenen Funktionen. Dabei entsteht ein *Call–Graph* – eine netzwerkartige Struktur, welche die Aufrufpfade und Abhängigkeiten von Funktion zu Funktion angibt. Die zur Laufzeit gewonnenen Erkenntnisse werden mit der statischen Architektur zu Gesamtbildern unterschiedlicher Komplexität kombiniert.

Tarantula [5] benutzt die Ergebnisse von Unit Tests [6]. Das ist zusätzlicher Code, welcher geschrieben wird, um definierte Funktionalitäten eines Modules automatisiert auf seine korrekte Funktionsweise zu überprüfen. Das Ergebnis dieser Überprüfung wird gemeinsam mit der statischen Architektur dargestellt. So können durch Codeänderungen induzierte Laufzeitprobleme in der Architektur besser gefunden werden.

Die Arbeit von Bohnet und Döllner [4] stellt einen Call–Graph in einer dreidimensionalen interaktiven Landschaft dar und ergänzt diesen mit dem zugehörigen Teil der Architektur. Beide Arbeiten verwenden eine geschickte Mischung aus der statischen Architektur und dynamischen Verhaltens zur Laufzeit.

Der Trend zu dreidimensionalen Darstellungen von Software ist ungebrochen. Eine Gesamtübersicht über aktuelle Verfahren der 3D Software Visualisierung findet sich in [11].

3. DESIGN

Unser Konzept geht von einem Funktionsaufruf aus, extrahiert aus dem entstehenden Call–Graph eine Baumstruktur und stellt diese als Variante eines Icicle Trees dar. Über die Verwendung von semitransparenten Würfelelementen in der Visualisierung können durch Veränderung des Blickwinkels unterschiedliche Betrachtungen gemacht werden. Die folgenden Schritte beschreiben die Herleitung dieses Konzepts im Detail.

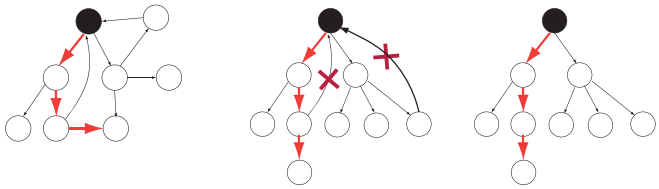


Abbildung 1. Reduktion des Call–Graphs zu einer Baumstruktur

Funktionsaufruf

Ein Funktionsaufruf ist der Eintrittspunkt zur Ausführung einer bestimmten Funktionalität der Software. Wir definieren diesen als gleichgültig mit dem Aufruf einer im Quellcode definierten Funktion oder Methode eines Objektes an. In der Vorstellung, dass jede Funktionalität auch höherer Abstraktion wiederum unter einem einzelnen Funktionsaufruf zusammengefasst werden kann, passen so auch komplexere Use–Case basierte Szenarien in dieses Raster.

Analyse Call–Graph

Während der Ausführung des Funktionsaufrufes werden weitere Funktionen aufgerufen. Diese sind von der Hauptfunktion abhängig. Aus dem spezifischen Aufrufmuster, oder in objektorientierter Terminologie, dem Kommunikationsmuster, erhalten wir dir Struktur eines gerichteten Graphen. Dieser wird als Call–Graph bezeichnet. Der Call–Graph enthält jede aufgerufene Funktion als Knoten. Rekursive Funktionsaufrufe führen dabei zu zyklischen Abhängigkeiten.

Da wir für die Darstellung der strukturellen Qualität nicht auf eine exakte Wiedergabe des Graphen angewiesen sind, komprimieren wir den Graphen auf eine Baumstruktur. Dazu erweitern wir die Kanten im Graphen um ein Gewicht, welches die Anzahl Aufrufe des entsprechenden Pfades zur Laufzeit angibt. Nun kann auf dem Graphen ein maximaler Spannbaum (analog dem Minimum Spanning Tree aus [8]) berechnet werden. Der so erhaltene Baum enthält implizit die kommunikative Charakteristik des Funktionsaufrufes. Abbildung 1 illustriert die Reduktion des Graphen zum Funktionsbaum.

Visualisierung des Baumes

Für die grafische Darstellung von Baumstrukturen können wir auf etablierte Algorithmen zurückgreifen. Eine Übersicht der Visualisierung von hierarchischen Strukturen findet sich in [14]. Für unsere Methode benutzen wir einen Icicle Tree In der Literatur [7] wird die Geschichte des Icicle Trees so

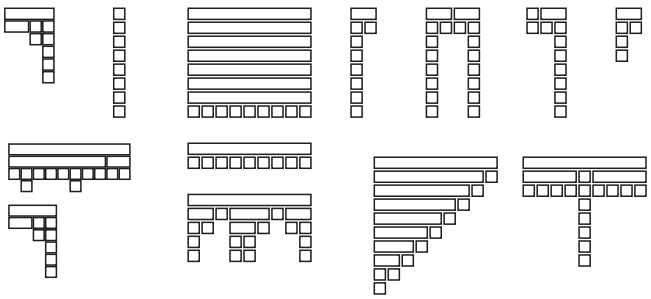


Abbildung 2. Verschiedene Varianten von Icicle Trees

beschrieben, dass dieser Baumstrukturen als Burgen (engl. Castles) visualisiert. Für unsere Darstellung funktioniert die Metapher der Burgen leider nicht: Burgen werden von unten gelesen. Unsere Blätter des Baumes sind aber die Funktionen, welche im Funktionsaufruf am tiefsten verschachtelt sind.

Gegenüber Kreisstrukturen hat der Icicle Tree den direkten Vorteil, dass er auf einem Raster basiert und dadurch alle Blätter des Baumes in der gleichen Grösse dargestellt sind. Das Raster lässt sich problemlos auch auf die dritte Dimension erweitern. Die Darstellungen der ganzen Funktionsbäumen sind in der Höhe und Breite vergleichbar. Abbildung 2 zeigt verschiedene Funktionsbäume und deren Darstellung als Icicle Tree.

Integration der statischen Architektur

Der Funktionsbaum stellt die Kommunikationspfade der Funktionen bezogen auf einen einzelnen Funktionsaufruf dar. Zur gezielten Integration der statischen Architektur, wählen wir eine Metrik, welche das Aufrufmuster bezogen auf die Gesamtarchitektur beschreibt. Diese Metrik ist die Instability (siehe oben).

Mit dem Wert der Instability modifizieren wir die Rasterelemente des Baumes: wir füllen die Elemente auf. Abbildung 3 zeigt links oben einen solchen modifizierten Baum.

3D, Versionen, Farbe

Als nächstes lesen wir den Baum spaltenweise und summieren die Höhe der gefüllten Elemente auf, wie in Abbildung 3 links unten dargestellt. Diese Leseart gibt uns die Verteilung der Stabilität über den ganzen Funktionsaufruf. Hat diese Verteilung Ausreisser, so könnte das ein Hinweis auf eine mögliche, architektonisch sinnvolle Aufteilung in mehrere Einzelfunktionen geben.

Eine Aufsummierung kann in dieser Visualisierung durch die Einführung von Transparenzen visuell erzeugt werden. Die natürliche Leseart des Baumes lässt sich dann dreidimensional verstehen.

Wir ersetzen die Basiselemente des Baumes deshalb durch Würfel. Aussen belassen wir Drahtwürfel als visuelle Marker für das Raster und setzen dann einen semitransparenten Würfel gemäss dem Wert der Instability in die Mitte. Abbildung 4 zeigt die so erzeugte Struktur eines einzelnen Funktionsaufrufs.

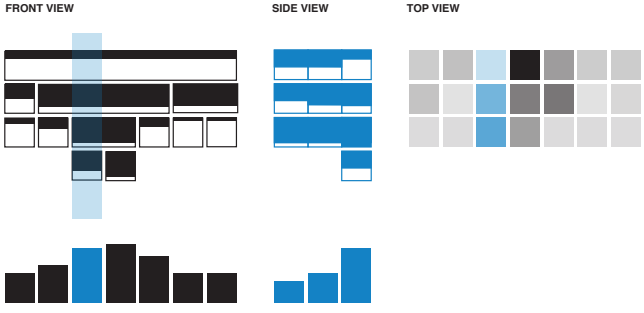


Abbildung 3. Modifizierter Icicle Tree mit semitransparenten Rasterelementen

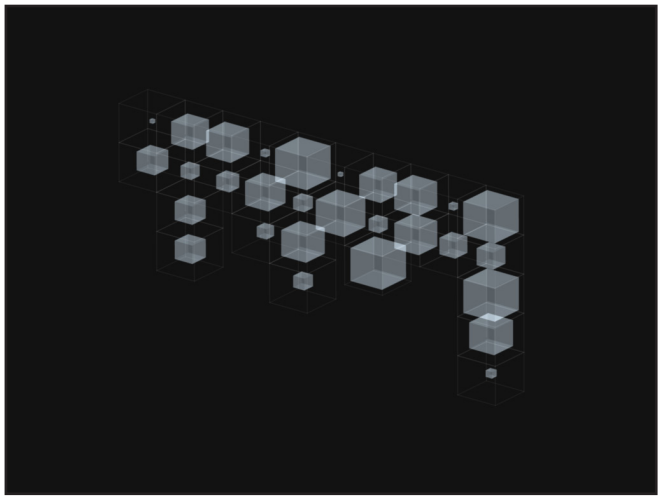


Abbildung 4. Einzelner Funktionsaufruf in 3D (isometrische Projektion)

Im nächsten Schritt ergänzen wir den Aufruf um dessen Versionsverlauf. Damit integrieren wir evolutive Aspekte in die Visualisierung, siehe Abbildung 5. Als Finish werden die Würfel farblich codiert. Dabei dient als Grundlage für die Farbgebund die Anzahl der Aufrufe. In Abbildung 9 sind deutlich einzelne *Hotspots*, also Funktionen mit vielen Aufrufen, ersichtlich.

Prototypische Umsetzung

Für die visuellen Aspekte dieses Konzeptes wurde ein einfacher Prototyp entwickelt. Dieser ist mit der 3D–Software Blender [3] entstanden. Ein parametrisierbares Python–Programm erzeugt zufällige Aufrufbäume in unterschiedlichen Grössenordnungen und Farbgebungen.

4. ERGEBNISSE

Die Abbildungen auf den folgenden Seiten zeigen ein Spektrum verschiedener Funktionsbäume, welche mit dem Prototyp erzeugt wurden. Es sind in den unterschiedlichen Ansichten jeweils verschiedene Merkmale ersichtlich. In Abb. 9 ist ein über den Versionverlauf zunehmender Hotspot erkennbar. Die Ansicht von vorne in Abb. 10 zeigt aufgrund des roten Balkens, der sich oben links befindet, dass die Funktion Kandidat für eine Teilung sein könnte. Ebenso ist erkennbar, dass die Hotspots vor allem in der neusten Version der Software auftauchen und daher ein evolutives Phänomen sind. Die relative Grösse der Funktionen im Hotspot deutet ausserdem an, dass sie eine hohe Instabilitätsmetrik haben und damit an vielen Orten der Gesamtarchitektur eingebunden sind. Aus diesem Grund könnte sich also eine Performance–Optimierung lohnen.

Diskussion

Grundsätzlich kann aus allen Bildern etwas abgelesen werden. Ob und wie sich diese Ergebnisse jedoch interpretieren lassen, ist schwer zu sagen. Hierfür müsste der Prototyp für die Arbeit auf realen Daten erweitert werden und die Ergebnisse im Detail evaluiert werden.

Im jetzigen Moment können die erreichten Darstellungen nur als visuelle Inspiration für weitere Experimente interpretiert

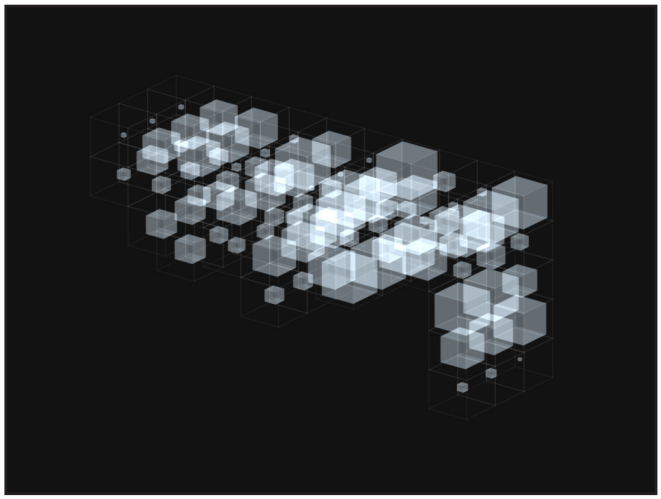


Abbildung 5. Drei Versionen im Versionsverlauf (von hinten nach vorne)

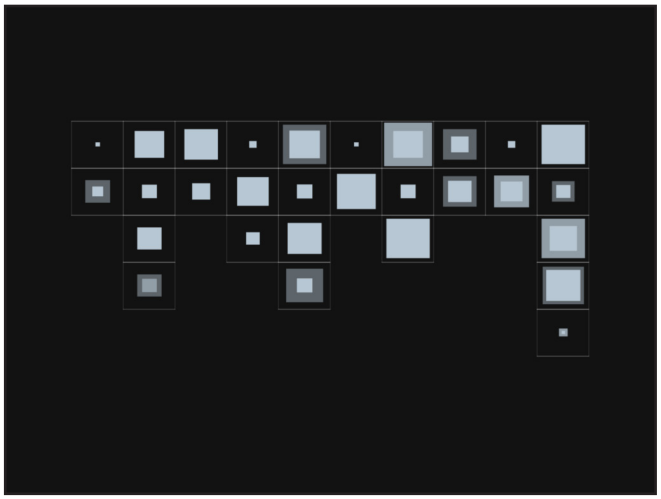


Abbildung 6. Ansicht von vorne (Überlagerung des Versionsverlaufs)

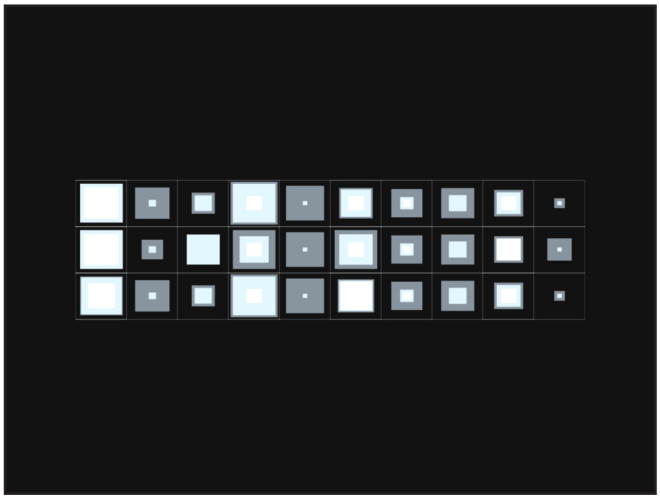


Abbildung 7. Ansicht von oben (Überlagerung der Ausführungspfade, Versionsverlauf von unten nach oben)

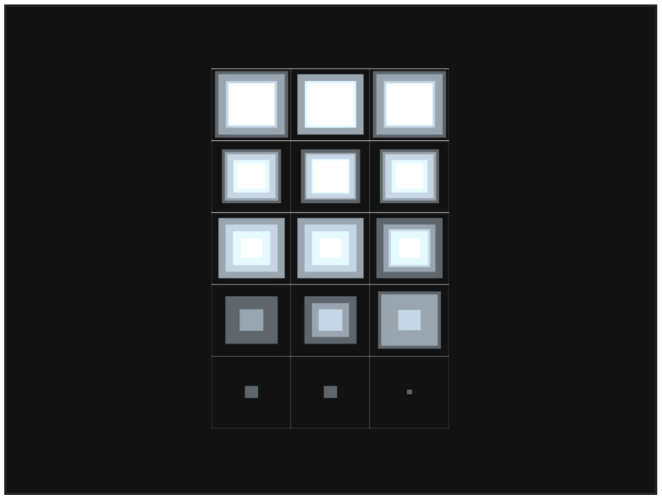


Abbildung 8. Ansicht von der Seite (Überlagerung der Funktionen gemäss Verschachtelungstiefe, Versionsverlauf von rechts nach links)

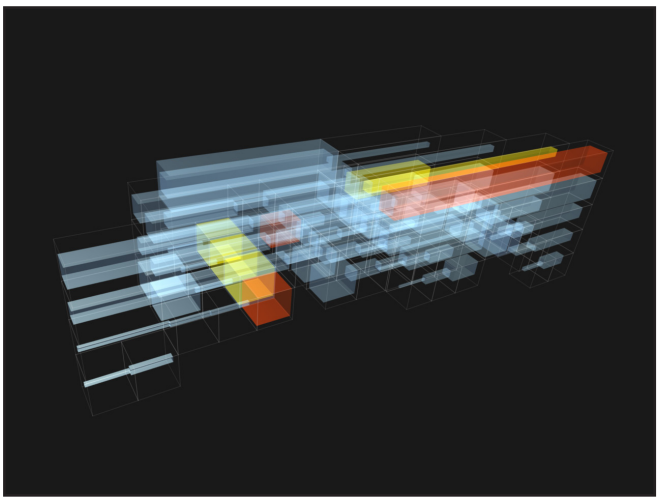


Abbildung 9. Darstellung von Hotspots

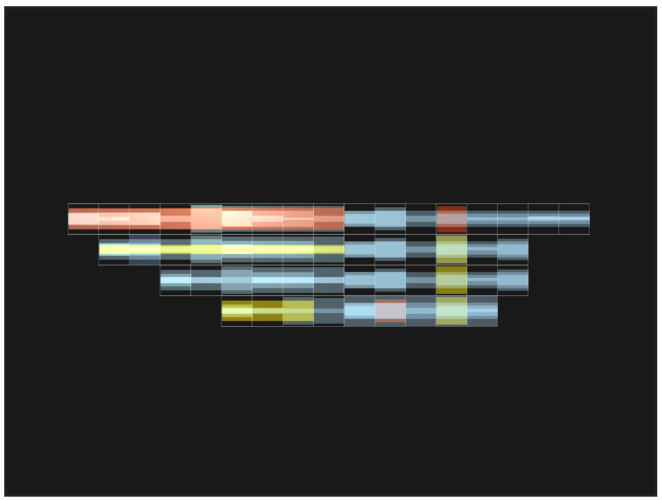


Abbildung 10. Entwicklung der Hotspots im Versionsverlauf (von unten nach oben)

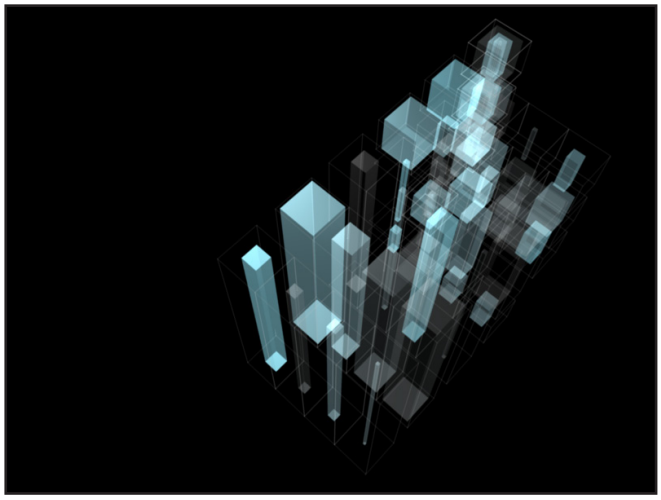


Abbildung 11. Einzelner Funktionsaufruf in 3D von unten (isometrische Projektion)

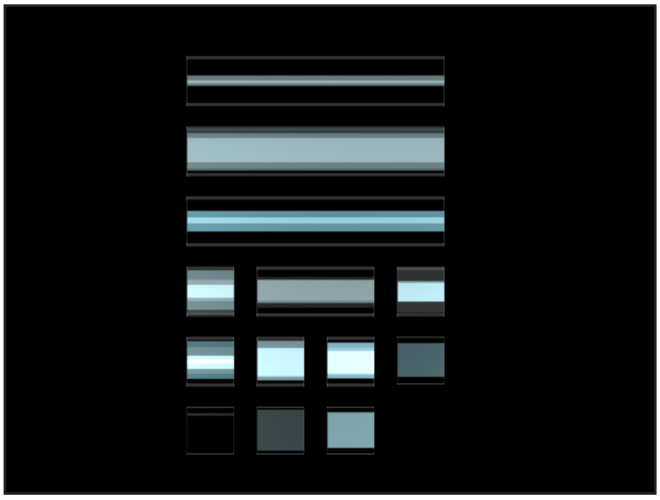


Abbildung 12. Seitliche Ansicht des Aufrufs aus Abbildung 11

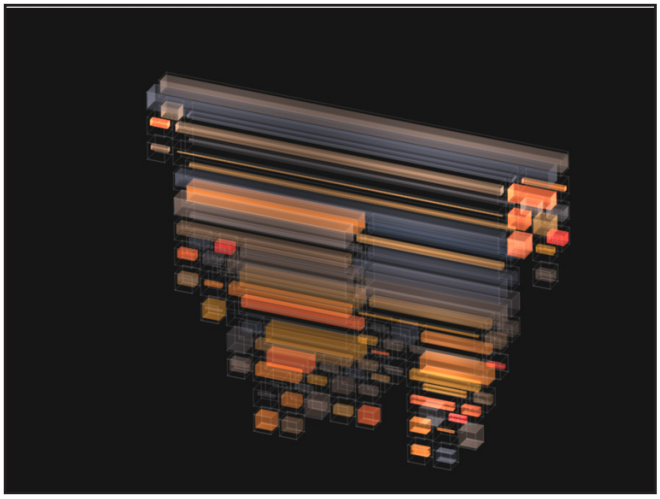


Abbildung 13. Variante der Farbgebung, grösserer Funktionsbaum

Abbildung 14. Ansicht von der Seite

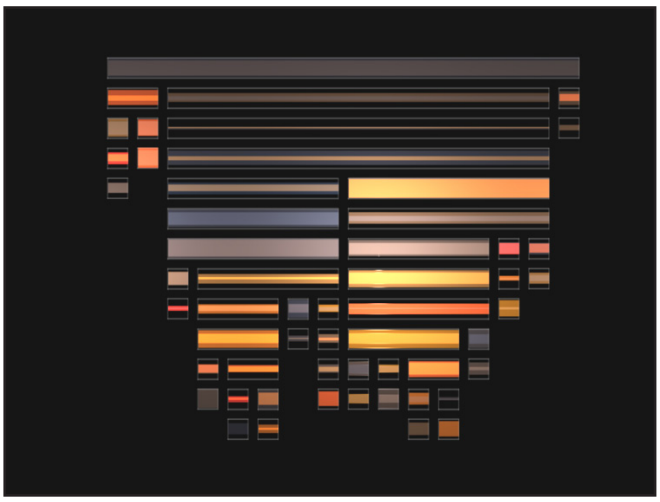


Abbildung 15. Ansicht von vorne

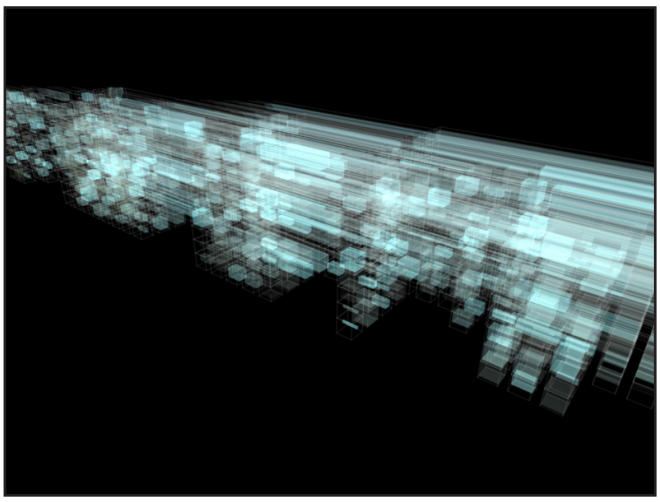
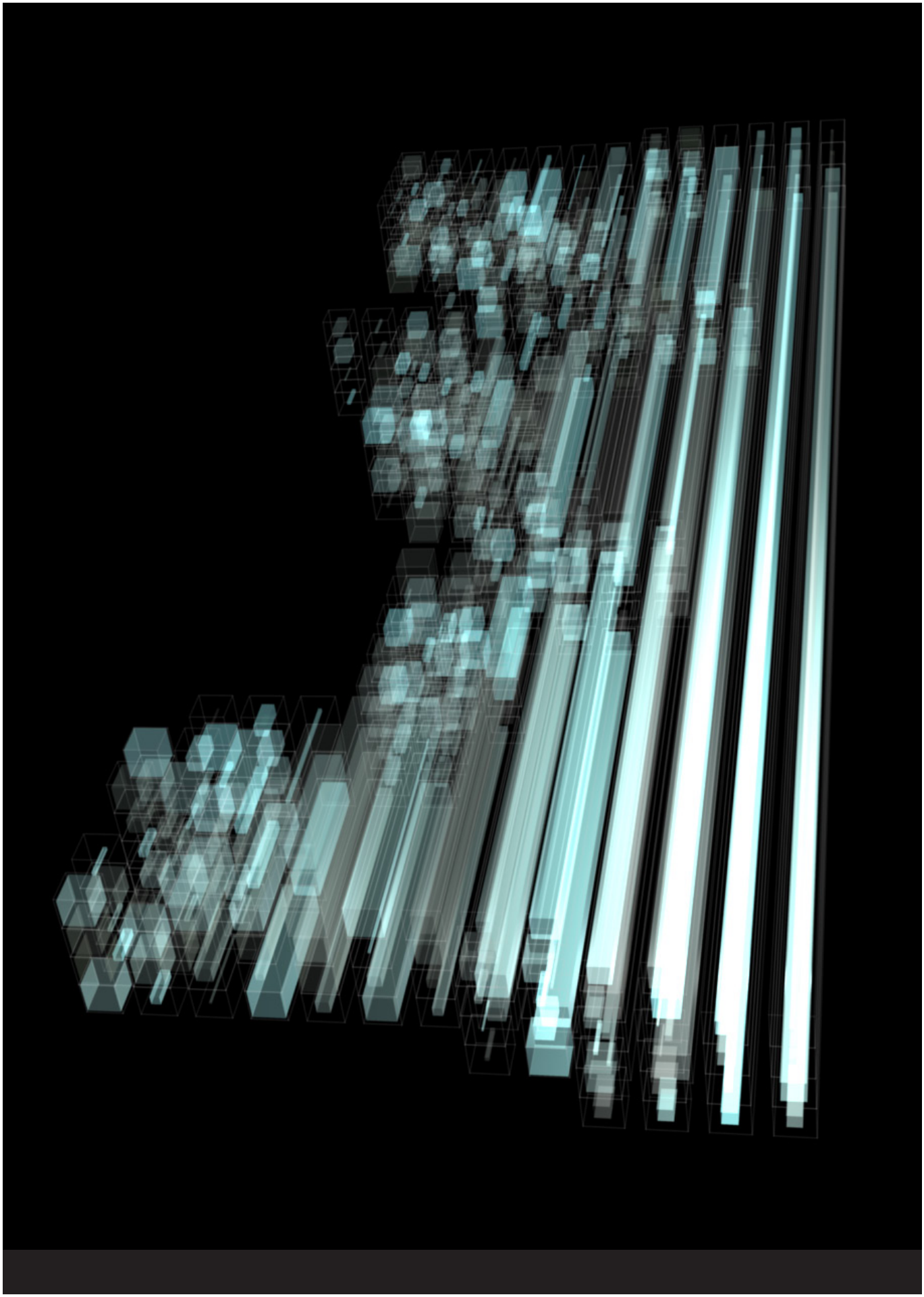


Abbildung 16. Darstellung grosser Funktionsbäume



werden. Insbesondere wurden auf der konzeptioneller Ebene viele Entscheidungen getroffen, die sich noch nicht im Prototyp wiederfinden und die jeweils einer seperaten vertieften Evaluation bedürften. Insbesondere die Herleitung des Funktionsbaumes über den maximalen Spannbaum eröffnet für sich alleine ein grosses Feld.

Interessant im Bereich der Visualisierung ist sicherlich der Aspekt der semitransparenten Rasterelemente, welche durch die visuelle Überlagerung der Struktur, verschiedene Lesemöglichkeiten mit sich bringt. Die Anwendung dieser Methode auf andere Baumstrukturen könnte Potential aufweisen.

In Bezug auf die Software–Visualisierung stellt sich auch noch die Frage der Vergleichbarkeit der einzelnen Funktionsbäume über mehrere Versionen. Da die Bäume jeweils zur Laufzeit entstehen, könnten diese bei jedem Aufruf sehr unterschiedliche Formen annehmen. Die Vergleichbarkeit und die charakteristische Struktur wäre in dem Fall nicht mehr gegeben. Als Lösungsansatz wäre das Vertauschen der statischen und dynamischen Werte zu überlegen. Ein Ansatz, der diese Arbeit näher an CodeCity heranrückt.

5. ZUSAMMENFASSUNG

Wir haben ein Konzept entwickelt, um Funktionsaufrufe als charakteristische räumliche Strukturen darzustellen. Bestimmte visuelle Aspekte haben wir mit einem Prototyp näher analysiert. Die Resultate zeigen, dass sich grundsätzlich spannende Bildwelten und Möglichkeiten in der Visualisierung ergeben. Insbesondere der Aspekt der visuellen Aufsummierung der Elemente und der mit dem Blickwinkel verbundenen interaktiven Erfahrbarkeit, eröffnet ein interessantes Feld.

Die Interpretierbarkeit der Strukturen im Gebiet der Software–Visualisierung erweist sich als schwierig, da zu viele Fragen offen sind und der entwickelte Prototyp nicht mit realen Datenquellen oder Zahlen arbeitet.

Das vorgestellte Konzept kann im Bereich der Visualisierung und Interaktion als erster Entwurf für weitere Arbeiten gesehen werden. Eine Entschlackung des Konzeptes durch Entfernung der disziplinbezogenen Aspekte ist jedoch erforderlich.

DANK

An Sandro Boccuzzo vom Institut für Informatik der Universität Zürich für die Einführung und Begleitung im Gebiet der Software Visualisierung.

LITERATUR

1. K. Beck: Simple Smalltalk Testing – With Patterns. Online verfügbar unter http://www.xprogramming.com/testfram.htm, zuletzt geprüft am 15.1.2009.

2. Boccuzzo, S. und Gall, H. C. (2007): Cocoviz: Supported cognitive software visualization. In: Proceedings of 14th Working Conference on Reverse Engineering (WCRE 2007), pages 273–274. IEEE Computer Society.

3. Blender Foundation: Blender 2.48a. Online verfügbar unter http://www.blender.org zuletzt geprüft am 15.1.2009.

4. Bohnet, J. und Döllner, J. (2006): Visual exploration of function call graphs for feature location in complex software systems. In: Proceedings of the 2006 ACM Symposium on Software Visualization (Brighton, United Kingdom, September 04 - 05, 2006). SoftVis `06. ACM, New York, NY, 95-104. Online verfügbar unter doi:10.1145/1148493.1148508, zuletzt geprüft am 15.1.2009.

5. Jones, J. A., Harrold, M. J., und Stasko, J. (2002): Visualization of test information to assist fault localization. In: Proceedings of the 24th international Conference on Software Engineering (Orlando, Florida, May 19 - 25, 2002). ICSE `02. ACM, New York, NY, 467-477. Online verfügbar unter doi:10.1145/581339.581397, zuletzt geprüft am 15.1.2009.

6. Gorman, J. (2006): OO Design Principles & Metrics. Online verfügbar unter http://www.parlezuml.com/metrics/OO%20Design%20Principles%20&%20Metrics.pdf, zuletzt geprüft am 15.1.2009.

7. Kleiner, B. und Hartigan, J. A. (1981): Representing points in many dimensions by trees and castles, Journal of the American Statistical Association.

8. Kruskal, J. (1956): On the shortest spanning subtree and the traveling salesman problem. In: Proceedings of the American Mathematical Society. 7 (1956), S. 48–50.

9. Metrics Eclipse Plugin. Online verfügbar unter http://metrics.sourceforge.net/, zuletzt geprüft am 15.1.2009.

10. Microsoft Code Analysis Team (2007): New for Microsoft Visual Studio 2008 – Code Metrics. Online verfügbar unter http://blogs.msdn.com/fxcop/archive/2007/10/03/new-for-visual-studio-2008-code-metrics.aspx, zuletzt geprüft am 15.1.2009.

11. Teyseyre R. und Campo, M. R. (2009): An Overview of 3D Software Visualization. In: IEEE Transactions on Visualization and Computer Graphics, Vol. 15, No. 1/2009.

12. Wettel, R. und Lanza, M. (2007): Program Comprehension through Software Habitability. In Proceedings of the 15th IEEE international Conference on Program Comprehension (June 26 - 29, 2007). ICPC. IEEE Computer Society, Washington, DC, 231-240. Online verfügbar unter doi:10.1109/ICPC.2007.30, zuletzt geprüft am 15.1.2009.

13. Wilkinson, L. (1982): An experimental evaluation of multivariate graphical point representations. In Proceedings of the 1982 Conference on Human Factors in Computing Systems (Gaithersburg, Maryland, United States, March 15 - 17, 1982). ACM, New York, NY, 202-209. Online verfügbar unter doi:10.1145/800049.801781.

14. Xu, K. (2007): Tree Drawing Algorithms and Visualization Methods. Online verfügbar unter http://www.cs.usyd.edu.au/~visual/comp4048/Tree_visualization.ppt, zuletzt geprüft am 15.1.2009.