Software-Visualisierung heute und morgen

Wie man sich einen Einblick in Software verschafft

Prof. Dr. Rainer Koschke, Universität Bremen, Axivion GmbH

Während ein Maschinenbauer an sein Werk treten und es sehen, fühlen, riechen und hören kann, entzieht sich Software der Sinneswahrnehmung ihrer Entwickler, weil Software immateriell ist. Nichtsdestotrotz müssen wir ihre oftmals komplexe innere Struktur und das Zusammenwirken ihrer Bestandteile durchschauen. Das Lesen des Quelltexts ist ab einem bestimmten Umfang nicht mehr möglich. Wir brauchen eine abstraktere Aufbereitung der Information. Software-Visualisierung ist die Wissenschaft der graphischen Repräsentation von Informationen über Software. Sie ist weit mehr als nur das Erzeugen von Pixeln. Die Interaktion in Form von Filtern, Suchen und Abfragen der dargestellten Daten ist integraler Bestandteil. Sie ist zudem eng verknüpft mit der Analyse von Software zur Datenextraktion. In der so genannten Visual Analytics hilft sie, die automatische Datenanalyse mit der menschlichen Fähigkeit zu ergänzen, Muster und Trends visuell auf einen Blick zu erfassen.

Dieser Beitrag beschreibt Techniken, Methoden und Werkzeuge der Software-Visualisierung. Sowohl der aktuelle Stand der Technik wird beleuchtet als auch aktuelle Trends aus der Forschung dargestellt und ein Ausblick auf die Zukunft gewagt. Folgende Aspekte werden behandelt – immer sowohl im Hinblick auf den heutigen Stand der Technik und den Stand der Wissenschaft: Was ist Software-Visualisierung? Was ist Visual Analytics? Was wird alles zu welchem Zweck visualisiert? Welche Arten von Visualisierungen und Interaktionsformen gibt es? Welche Eigenschaften menschlicher Wahrnehmung sind wie zu berücksichtigen? Welche Rolle spielen Metaphern? Wie kann ich meine eigene Software-Visualisierung selbst implementieren?

Was ist Software-Visualisierung und wozu ist das gut?

Bei vielen Wartungsproblemen kann man nicht vorab eine präzise Frage formulieren beziehungsweise einen Algorithmus angeben, der eine präzise Antwort auf die Frage liefern kann. Bei vielen
Wartungsproblemen muss man stattdessen eher explorativ vorgehen. Selbst wenn man präzise Fragen und Algorithmen hat, kann es dann immer noch geschehen, dass die Fülle der Resultate erdrückend ist. Für exploratives Vorgehen und die Untersuchung großer Datenmengen bieten sich Visualisierungen an, die dem menschlichen Betrachter helfen können.

Price, Baecker und Small beschreiben Software-Visualisierung wie folgt:

> *"Software Visualization is the use of the crafts of typography, graphic design, animation, and cinematography with modern human-computer interaction and computer graphics technology to facilitate both the human understanding and effective use of computer software.*
> *Program visualization is the visualization of the actual program code or data*

structures in either static or dynamic form."

Die Notwendigkeit von Software-Visualisierung wird von Thomas Ball herausgestellt:

> *"Software is intangible, having no physical shape or size. Software visualisation tools use graphical techniques to make software visible by displaying programs, program artifacts and program behaviour."*

Software-Visualisierung ist ein Spezialgebiet der allgemeinen Informationsvisualisierung. Informationsvisualisierung beschäftigt sich mit der Darstellung von Informationen, so dass sie menschlichen Betrachtern besser zugänglich ist. Viele Erkenntnisse in diesem Bereich lassen sich auf Software-Visualisierung im Speziellen übertragen. Dennoch ist es notwendig, die Besonderheiten des visualisierten Gegenstands – in diesem Falle also Software – zu berücksichtigen. Software ist immateriell, wie bereits erwähnt. Es gibt somit keine natürliche Darstellungsform. Software hat sowohl Struktur als auch Verhalten und beides sollte visualisiert werden können. Software ist mehrschichtig, komplex und detailreich. Graphische Repräsentationen müssen mit textuellen Darstellungen verknüpft werden, da die meisten Programmiersprachen textuell sind und es letztlich die Programmtexte sind, die die vollen Details enthalten und die durch Visualisierung abstrahiert werden soll. Die Änderungen müssen im Quelltext gemacht werden. Die Visualisierung ist ein Mittel, um dies effizient und effektiv erreichen zu können.

Was wird alles zu welchem Zweck visualisiert?
In der Software-Visualisierung werden Informationen über Software dargestellt. Dazu gehören sowohl statische als auch dynamische Aspekte. Statische Informationen sind solche, die direkt durch Analyse von Dokumenten gewonnen werden können, ohne das Programm ausführen zu müssen. Zu den Dokumenten gehören nicht nur der Quellcode, sondern auch Beschreibungen der Architektur und der Testfälle sowie die Dokumentation. Metriken zur Größe und Komplexität von Funktionen beispielsweise sind klassische Daten, die durch eine statische Analyse gewonnen werden. Dynamische Analysen ermitteln ihre Ergebnisse durch Ausführung des Programms. Beispiele hierzu sind der Speicherverbrauch, Laufzeiten oder konkrete Werte, die ein Programm für eine bestimmte Eingabe liefert.

Das Ziel der Software-Visualisierung ist die Reduktion der Komplexität für den Betrachter, verschiedene Sichten zu bieten und Inferenzen des menschlichen Betrachters zu unterstützen. Dabei ergeben sich verschiedene Herausforderungen. Zum einen muss eine Visualisierung skalieren, da wir es in der Wartung meist mit sehr großen Programmen zu tun haben. Was für die Darstellung eines Miniprogramms gut aussieht, ist nicht notwendigerweise auch anwendbar für sehr große Programme. Zum andern muss eine passende Form der Visualisierung gefunden werden, die alle für ein Problem relevanten Aspekte erfasst und von allen irrelevanten abstrahiert. Die Güte einer Software-Visualisierung ist somit grundsätzlich abhängig von der konkreten Aufgabe, die der Entwickler verfolgt. Ob Detailinformationen oder grobe Übersichten notwendig sind, bestimmt darüber, welche Art von Software-Visualisierung in Frage kommt. Unter den vielen möglichen Visualisierungen müssen dann die geeigneten für die jeweilige

Aufgabe ausgewählt werden. Dazu ist es hilfreich, eine Übersicht über Visualisierungen zu haben. Dieser Beitrag gibt eine solche Übersicht.

Was sind Software-Analytics und Visual Analytics und wie hängen sie mit Software-Visualisierung zusammen?

Software-Visualisierung ist verwandt mit der so genannten Software-Analytics. *Software-Analytics* ist die Wissenschaft zur Entdeckung und Kommunikation von bedeutungsvollen Mustern in Software-Daten. Sie verwendet dazu Methoden der Statistik, Data-Mining-Techniken, maschinelles

Lernen und Visualisierung. Software-Visualisierung ist darin eine Stütze. Software-Analytics geht aber darüber hinaus, indem Methoden angewandt werden, die Daten gewinnen und Zusammenhänge und Muster erkennen. *Visual Analytics* bezogen auf Daten über Software und ihren Entwicklungsprozess kann als ein Teilgebiet der Software-Analytics betrachtet werden. Wie auch bei der Software-Analytics ist das Ziel der Visual Analytics, Erkenntnisse aus großen und komplexen Datensätzen zu gewinnen. Hierzu werden jedoch nicht nur klassische Methoden der Statistik, Data-Mining-Techniken und maschinelles Lernen eingesetzt. Der Ansatz kombiniert überdies diese automatischen Datenanalysetechniken mit der menschlichen Fähigkeit, schnell Muster oder Trends visuell erfassen zu können. Daten werden vom menschlichen Betrachter visuell exploriert, um Erkenntnisse zu gewinnen. Visualisierung spielt hierbei also eine zentrale Rolle und dient nicht nur der Darstellung der Endergebnisse wie in der klassischen Software-Analytics.

Welche Eigenschaften menschlicher Wahrnehmung sind wie zu berücksichtigen?

Software-Visualisierungen müssen mit begrenzten Ressourcen auskommen. Die Ressourcen sind sowohl durch Werkzeuge und Hardware vorgegeben, wie beispielsweise die Auflösung und Größe der Anzeigefläche. Aber auch die menschlichen kognitiven Fähigkeiten haben ihre Grenzen, über die sich eine Software-Visualisierung nicht hinwegsetzen kann. Die menschliche Wahrnehmungsfähigkeit wie Scharf- und Farbsehen müssen – wie bei jedem anderen Interaktionsdesign der Mensch-Maschine-Kommunikation auch – mit berücksichtigt werden. Software-Visualisierung ist aber möglicherweise noch schwieriger als Visualisierungen anderer Dinge, weil es im Unterschied zu anderen Domänen in der Software-Entwicklung weniger Konventionen für Visualisierungen gibt, also zum Beispiel wie Farben und Symbole zu interpretieren sind. In der Software-Wartung existierender Programme haben wir es darüber hinaus mit einem besonders umfangreichen und komplexen Informationsraum zu tun.

Die visuelle Wahrnehmung unterliegt bestimmten kognitiven Gesetzmäßigkeiten, die fest in unserem Gehirn verankert sind. Die kognitive Psychologie beschäftigt sich mit diesen im Allgemeinen. Die Gestaltpsychologie hat beispielsweise Ordnungsprinzipien bei der Wahrnehmung insbesondere unbewegter zweidimensionaler Bilder identifiziert. Dazu gehören etwa das Gesetz der Nähe, nach dem Elemente mit geringen Abständen zueinander als zusammengehörig wahrgenommen werden, oder das Gesetz der Ähnlichkeit, nach dem einander ähnliche Elemente eher als zusammengehörig erlebt werden als einander unähnliche. Viele visuelle Eindrücke gelangen gar nicht erst ins Bewusstsein beziehungsweise werden schon vorher verarbeitet. Die so genannte

präattentive Wahrnehmung ist eine vorbewusste, unterschwellige Wahrnehmung von Sinnesreizen. Ein Reiz wird zwar vom Nervensystem einer Person wahrgenommen und löst dort auch einen Effekt aus, dringt jedoch nicht bis ins Bewusstsein. Dies beschleunigt einerseits die Wahrnehmung, weil diese Prozesse parallel und automatisiert ablaufen, kann jedoch auch dazu führen, dass wichtige Informationen schon ausgefiltert werden und das Bewusstsein gar nicht „die volle Wahrheit“ erfährt. Im Zusammenhang von sich verändernden visuellen Sinneseindrücken ist die so genannte Veränderungsblindheit des Menschen eine weitere Gefahr für Unvollständigkeiten und visuelle Trugschlüsse. Die Veränderungsblindheit bezeichnet ein Phänomen der visuellen Wahrnehmung, bei dem Änderungen in einer visuellen Szenerie vom Betrachter nicht wahrgenommen werden, obwohl die Änderungen gravierend sind. In einem klassischen Experiment, bei dem die Betrachter mit der Aufgabe beschäftigt werden zu zählen, wie oft eine Gruppe von Menschen sich einen Ball zuwirft, wandert zwischendurch eine Person durch das Bild, die als Gorilla verkleidet ist. Ein Großteil der Betrachter nimmt diesen Gorilla gar nicht wahr, obwohl er deutlich zu sehen ist. Zu sehr sind die Betrachter mit ihrer Aufgabe beschäftigt. Ihr Fokus auf die gestellte Aufgabe ist zugleich eine Verengung ihrer Sinneswahrnehmung. Ein ähnliches Phänomen ist die Unaufmerksamkeitsblindheit. Dies ist die Nichtwahrnehmung von Veränderungen von Objekten, die nicht im Fokus stehen. Werden bei zwei Bildern der gleichen Szenerie marginale und für den Betrachter unbedeutende Veränderungen vorgenommen, fällt dies dem Betrachter bedingt durch die eingeschränkte Verarbeitungskapazität des menschlichen Gehirns nichts ins Auge.

Wer also Visualisierungen verwendet, um die menschliche Fähigkeit auszunutzen, rasch Muster zu erkennen, der muss sich gewahr sein, dass das menschliche Gehirn erst selektieren muss, welche Informationen relevant sind und welche weniger. Es muss einer Visualisierung gelingen, die menschliche Aufmerksamkeit auf relevante Sachverhalte zu lenken, damit diese bewusst wahrgenommen werden können.

Welche Arten von Visualisierungen und welche Interaktionsformen gibt es?

Graphen sind eine weit verbreitete Form der Visualisierung in der Wartung. Sie bieten sich in natürlicher Weise an für Abhängigkeiten zwischen Software-Teilen und andere binäre Relationen. Auch die meisten UML-Diagrammarten wie Klassendiagramme oder Kollaborationsdiagramme sind letzten Endes Graphen. Ein Beispiel für die Visualisierung von Klassendiagrammen ist in Abbildung 1 gezeigt. Das UML-Diagramm selbst ist in der rechten oberen Ecke zu finden. Darum herum findet man alternative Sichten. Unterhalb des Diagramms wird Quellcode angezeigt. Die Darstellung weist eine Reihe wichtiger Eigenschaften auf. Es stellt beispielsweise verschiedene Sichten dar, abstraktere und detaillierte. Es ist bei Software-Visualisierungen im Allgemeinen auch notwendig, einen Bezug von der Visualisierung zum Quellcode herzustellen, wie es in dieser Darstellung geschieht. Links vom UML-Diagramm sieht man eine Darstellung der Hierarchie. Auch dies ist typisch, weil große Systeme hierarchisch strukturiert sind. Unter der Hierarchie findet man die Darstellung von Attributen eines ausgewählten Elements des UML-Diagramms. Hier lassen sich Details abfragen.

Diese Visualisierung unterstützt das von Shneiderman vorgeschlagene Vorgehen, das

jede Visualisierung nach anstreben sollte (so genanntes „Mantra der Informationsvisualisierung“): Zuerst soll eine Übersicht geboten werden. Dann kann man Informationen suchen und filtern. Schließlich erhält man Details auf Anfrage.

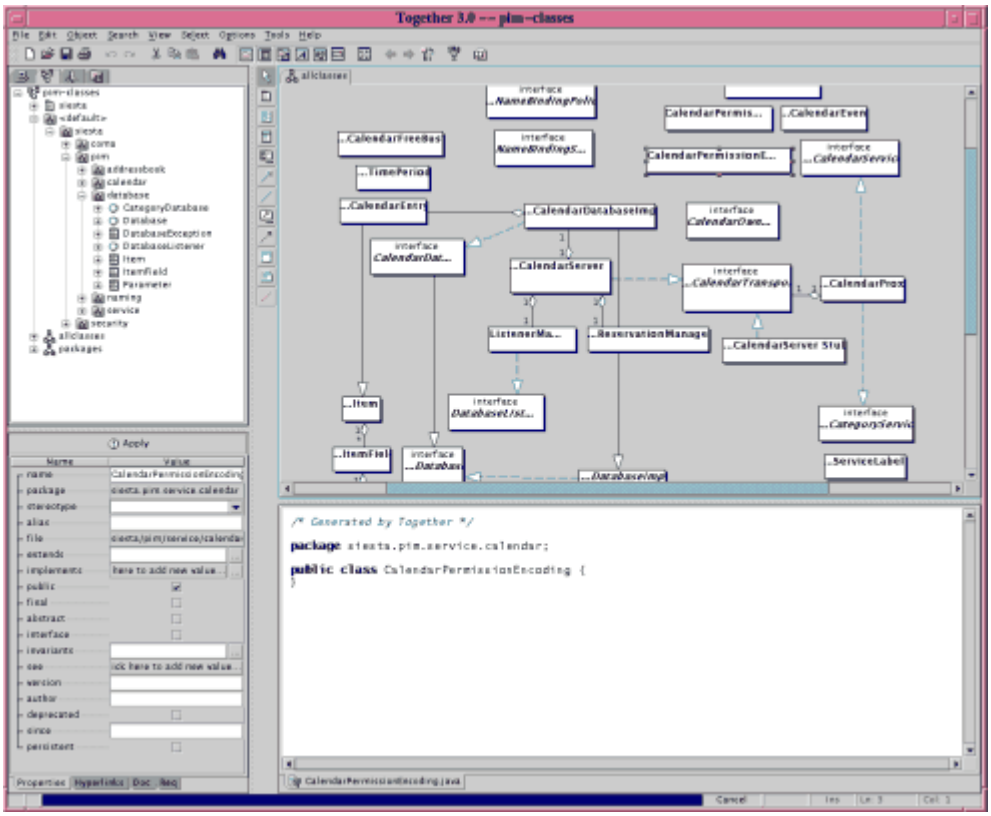

Abbildung 1: Klassendiagramme im UML-Werkzeug *Together*

UML-Werkzeuge können objektorientierte Konzepte mit graphischen Mitteln der UML darstellen. Was sich durch UML-Konzepte nicht ausdrücken lässt, kann man nicht damit visualisieren. Auch die Perspektiven, aus denen die Software dargestellt wird, sind vorgegeben und vom Benutzer meist nicht selbst frei festzulegen beziehungsweise zu erweitern.

Aus diesen Gründen ist es wünschenswert, generischere Darstellungsmöglichkeiten bereitzustellen, das heißt, alternative visuelle Sprachen von Nutzern selbst bestimmen zu lassen. Generische Darstellungen von Graphen und Möglichkeiten zur Erweiterung werden beispielsweise von dem in den Neunzigerjahren von Hausi Müller entwickelten Grapheneditor Rigi angeboten. Rigi wurde für die Visualisierung von Software mit Hilfe getypter Graphen entwickelt und ist eines der frühesten Werkzeuge seiner Art, das viele andere Werkzeuge maßgeblich beeinflusst hat. Es wird mittlerweile nicht mehr entwickelt, hat aber die Entwicklung vieler späterer Werkzeuge inspiriert.

Eine Darstellung eines Abhängigkeitsgraphen mit Rigi ist in Abbildung 2 gezeigt. Es zeigt Zusammenhänge von Funktionen und Datentypen. Die Typen der Knoten werden durch verschiedene Ikone und Farben unterschieden. Die Farbe der Kanten drückt deren Semantik aus, hier beispielsweise wird zwischen Funktionsaufrufen und Verwendungen

von Datentypen in der Signatur von Funktionen farblich differenziert.

Die Graphen, die Rigi verarbeiten kann, sind generisch. Das heißt, ein Verwender von Rigi kann die Typen der Knoten und Kanten selbst festlegen und Attribute frei spezifizieren. Generische Filter und Navigationsmöglichkeiten stehen für den Endanwender zur Verfügung. Zur Erweiterbarkeit bietet Rigi eine Scripting-Schnittstelle, die den Graphen abfragen und manipulieren kann. Auf diese Weise können eigene Sichten geschaffen werden. Der graphische Editor ist somit programmierbar.

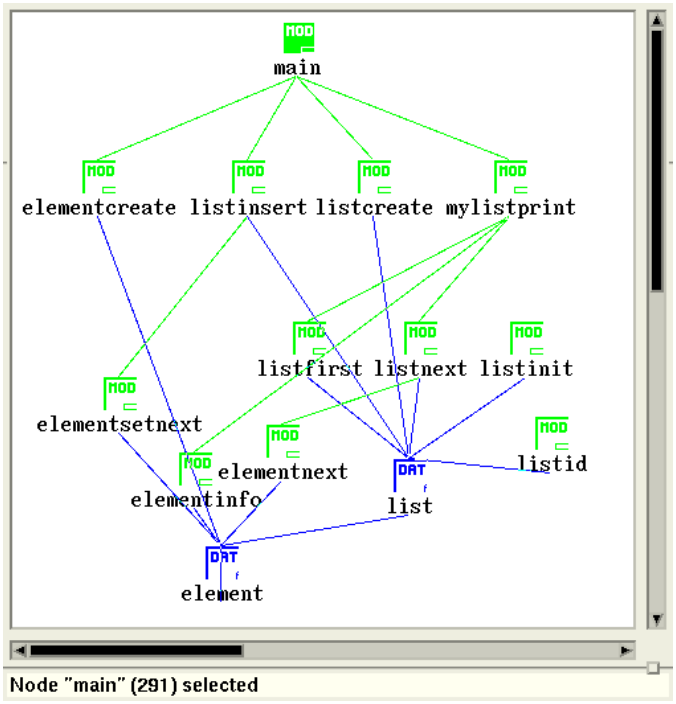

Abbildung 2: Visualisierung mit Rigi; die blauen Knoten stellen Typen dar, die grünen Funktionen; grüne Kanten repräsentieren, dass ein Typ in der Signatur einer Funktion auftritt, blaue Kanten, dass eine Funktion eine andere aufruft.

Die Graphen in Rigi sind hierarchisch. Das heißt, ein Knoten kann anderen Knoten enthalten. Das Zusammenfügen von Knoten kann entweder interaktiv durch den Benutzer vorgenommen werden oder aber programmatisch über die Scripting-Schnittstelle. Auf diese Weise kann man große unübersichtliche Graphen in Gruppen zusammenfassen. Das Kriterium für die Gruppierung ist anwendungsspezifisch und wird vom Benutzer beigesteuert. Die Forschung zum so genannten Software-Clustering beschäftigt sich mit dem automatisierten Gruppieren von Software-Einheiten. Abbildung 3 zeigt das Ergebnis eines solchen automatisierten Clusterings. Hier wurden die Funktionen mit den Datentypen, die in ihrer Signatur auftauchen, zu abstrakten Datentypen zusammengefasst.

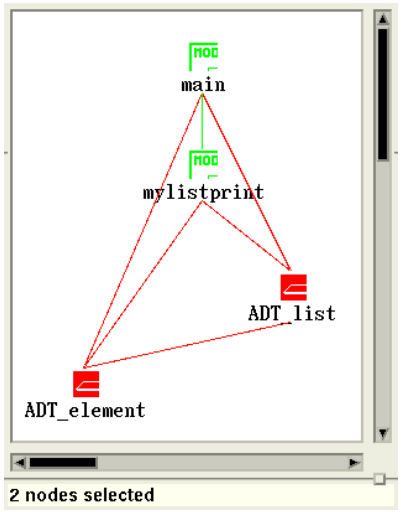

Abbildung 3: Visualisierung mit Rigi mit zusammengesetzten Knoten.

Eine Alternative zu Graphen zur Darstellung binärer Relationen sind **Adjazenzmatrizen**. Ein Aufrufgraph kann zum Beispiel wie in Abbildung 4 gezeigt dargestellt werden. An den beiden

Achsen sind alle Funktionen aufgeführt. An der Koordinate wird ein Punkt gesetzt, wenn die Funktion an der x-Achse eine andere Funktion an der y-Achse aufruft. Rekursive Aufrufe schlagen sich somit als ein Punkt auf der Diagonalen nieder. Der Punkt kann eingefärbt werden, um zusätzliche Information unterzubringen, zum Beispiel die Häufigkeit eines Aufrufs.

Die Vorteile dieser Darstellung sind ihre Einfachheit sowie Skalierbarkeit in Bezug auf den für die Darstellung notwendigen Rechenaufwands. Das automatische Layouting großer dichter Aufrufgraphen ist sehr rechenintensiv, während eine Adjazenzmatrix schnell dargestellt werden kann. Auch die Übersichtlichkeit ist bei Adjazenzmatrizen höher als bei dicht besetzten Graphen. Ihr Nachteil ist jedoch, dass Wege nicht zu erkennen sind. Hier sind Graphen deutlich überlegen. Zudem ist die Skalierbarkeit für Graphen mit sehr vielen Knoten in der Ausnutzung des verfügbaren Platzes für die Darstellung nicht gegeben. Graphen sind häufig dünn besetzt, wodurch der Platz nicht effizient ausgenutzt wird. Lange X- und Y-Achsen machen Schiebebalken oder andere Interaktionsformen notwendig, so dass man nicht immer alles auf einen Blick sieht.

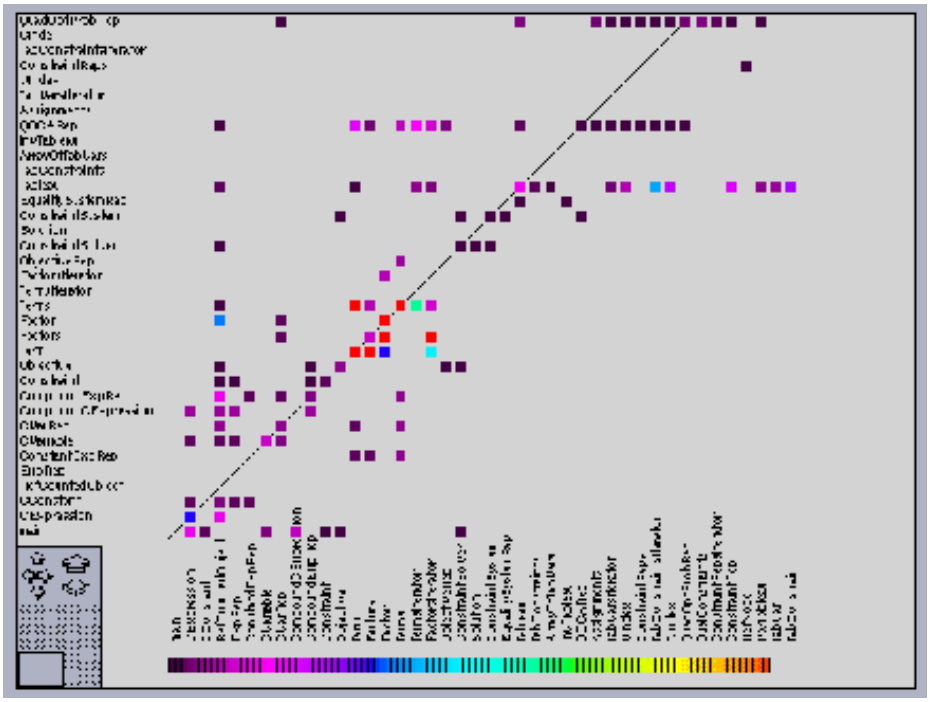

Abbildung 4: Aufrufgraph in Form einer Adjazenzmatrix.

Wenn es um Abfolgen geht (zum Beispiel bei den Ergebnissen dynamischer Analysen) kann man **Sequenzdiagramme** verwenden, die auch Teil der UML sind. Eine Erweiterung von UML-Sequenzdiagrammen durch Farben und anderes bietet JInsight. Jeder vertikale Streifen repräsentiert ein Objekt (vergleiche Abbildung 5). Die Farbe des Streifens kodiert die Klasse, zu der das Objekt gehört. Die Zeit schreitet wie bei Sequenzdiagrammen üblich von oben nach unten fort. Aufrufe sind durch Kanten zwischen Objekten dargestellt und haben die aufgerufene Methode als Label. Die zeitlichen Bereiche, in denen ein Objekt tatsächlich aktiv ist (d.h. nicht nur eine andere Methode aufruft sondern tatsächlich selbst rechnet), sind im Streifen etwas heller dargestellt.

Sequenzdiagramme, die tatsächliche Abläufe, die mittels einer dynamischen Analyse eines realistischen Programms gewonnen wurden, sind sehr lange. Häufig wiederholen sich darin Teilsequenzen, wenn diese etwa im Programm in einer Schleife ausgeführt werden. Würde ein langlaufendes Programm auf diese Weise dargestellt werden, würde man sich mit einem Schiebebalken sehr lange über die angezeigte Sequenz bewegen müssen. JInsight erlaubt es deshalb, wiederkehrende Aufrufmuster automatisch zu erkennen und darzustellen. Ähnliche, aber leicht unterschiedliche Muster können dabei übereinandergelegt werden, um so die Unterschiede darzustellen. Hier sieht man erneut, wie wichtig Vorverarbeitungsschritte in Form automatisierter Analysen sind, damit ein Bild lesbar und nützlich wird.

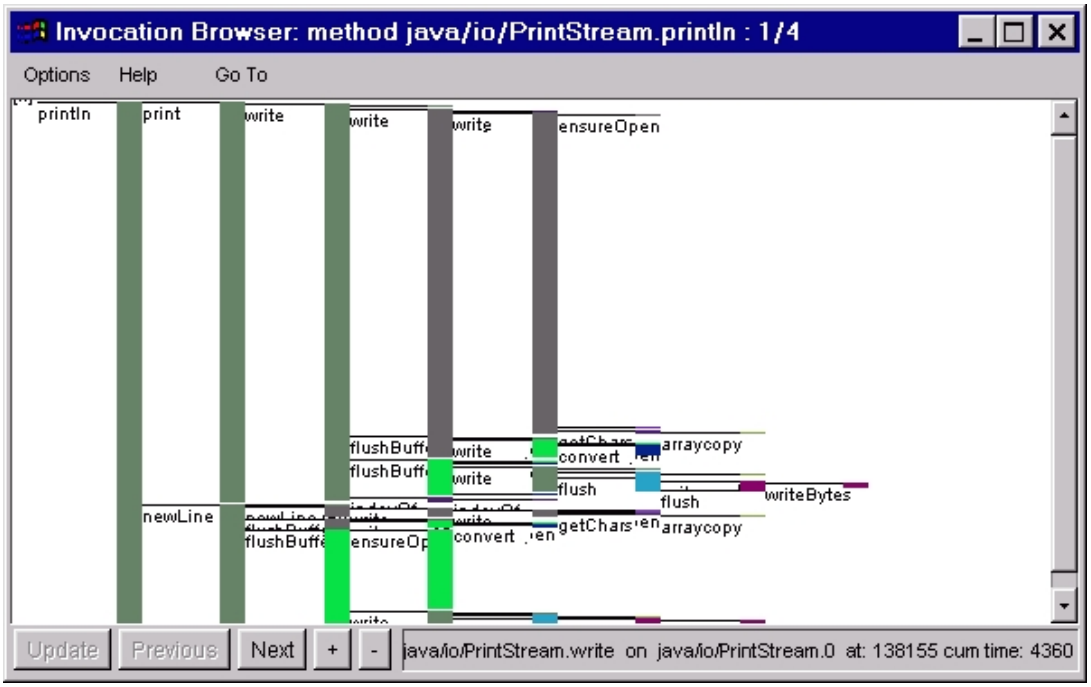

Abbildung 5: Darstellung von Sequenzdiagrammen mit JInsight.

Wenn es um die Visualisierungen von Metriken geht, kann man auf einen großen Satz klassischer Visualisierungen zurückgreifen. Dazu gehören **Tortendiagramme**, **Funktionsgraphen**, **Box-Plots**, **Balkendiagramme** und viele andere. Wichtig ist bei der Visualisierung, dass man die dargestellten Metriken sehr einfach mit anderen Visualisierungen der Software in Verbindung bringen kann. Wie wir bereits am Beispiel des Eingangs gezeigten UML-Werkzeugs gesehen haben, wird Software meist durch verschiedene Sichten unterschiedlichen Abstraktionsgrads abgebildet. Abbildung 6 zeigt auf der linken Seite einen Aufrufgraphen. Auf der rechten Seite sehen wir mehrere Koordinatensysteme. Jede Funktion im Aufrufgraphen wird in jedem Koordinatensystem als Punkt dargestellt. Die Koordinatensysteme beschreiben unterschiedliche Metriken zu verschiedenen Aspekten der Software. Dazu gehören beispielsweise statische Code-Metriken, wie die Anzahl von Kommentaren oder Bedingungen, dynamische Metriken wie zum Beispiel die Testabdeckung oder die Laufzeit, aber auch Metriken zu Aspekten des Entwicklungsprozesses, wie die Häufigkeit von Änderungen an den Funktionen oder die Anzahl der verschiedenen Entwickler, die die Funktionen geändert haben.

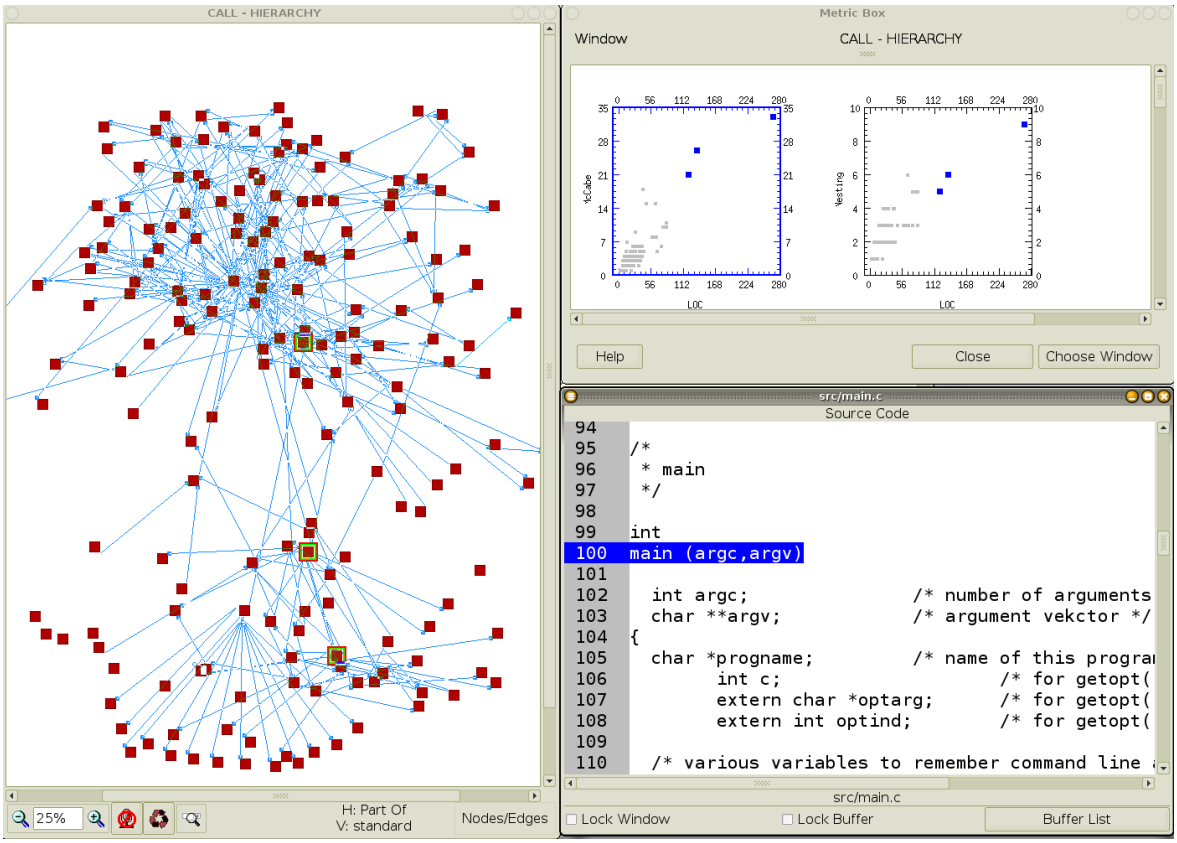

Abbildung 6: Erkennung von Bad-Smells durch Metriken

Der Bezug zwischen den Koordinatensystemen und dem Aufrufgraph wird hergestellt, indem die Punkte selektiert werden. Wird eine Funktion im Aufrufgraph selektiert, dann wird der entsprechende Punkt in allen Koordinatensystemen farbig hervorgehoben und umgekehrt.

Wenn es um die Darstellung von Hierarchien geht, deren Blätter mit Metriken assoziiert sind, kann man sich der so genannten **Tree-Maps** bedienen. Hier wird eine Baumstruktur in die Ebene projiziert. Die Fläche, die ein Blatt einnimmt, ist proportional zu ihrem Metrikwert. Um gleichzeitig die Hierarchie abzubilden, werden alle Nachfolger in einem Teilbaum räumlich geschachtelt dargestellt. Die Hierarchiegrenzen werden mit einem Rahmen versehen.

Abbildung 7 hilft, das Vorgehen zu erläutern. Zunächst beginnt man mit einer quadratischen Fläche, auf die der Baum projiziert werden soll. Dann aggregiert man die Metrikwerte durch Summenbildung von den Blättern zu den Vorgängern im Baum. Der Knoten X hat dann den Wert 99, der Knoten Z den Wert 50, der Knoten Y den Wert 100 und die Wurzel W schließlich den Wert 199. Die Flächen werden nun von der Wurzel ausgehend proportional auf die Nachfolger im Baum verteilt. Die beiden Knoten X und Y haben nahezu denselben Anteil 99/199 beziehungsweise 100/199. Diese Werte werden rekursiv auf die Nachfolger verteilt. Die Knoten a, b und c erhalten alle den gleichen Anteil, der ihrem gemeinsamen Vorgänger X zugesprochen wurde, nämlich jeweils ein Drittel von 99/199. Die Knoten Z und d erhalten jeweils die Hälfte von 100/199. Für die

Nachfolger von Z wird dieser Anteil abermals halbiert.

Damit die Schachtelung des Baumes deutlich wird, werden die Flächen alternierend in der Baumtiefe vertikal und dann horizontal verteilt. Schließlich ergibt sich dann die Fläche, die rechts neben dem Baum in Abbildung 7 gezeigt ist.

Ein realistischeres Beispiel ist in Abbildung 8 dargestellt. Es stellt Redundanzen in Form von dupliziertem Code in einem sehr großen realen Programm dar, das in einem industriellen Unternehmen entwickelt wurde. Die Struktur der Tree-Map gibt die Quelltextdateien und ihre Organisation in der Verzeichnisstruktur wieder. Die Flächen sind proportional zur relativen Größe der Dateien, gemessen in Code-Zeilen. Der Grad der Redundanz wird durch die rote Einfärbung abgebildet; je redundanter eine Datei ist, desto dunkelroter ist die Fläche. Die blaue Fläche ist die Datei, die vom Nutzer im Augenblick selektiert wurde. Alle anderen Teile, die Redundanzen mit der selektierten Datei haben, sind grün eingefärbt. Kanten verbinden die grünen Teile mit der selektierten blauen Fläche. Der Grad von hell- bis dunkelgrün zeigt wie im Falle der unverbundenen Flächen den Grad der Redundanz an – mit der blauen oder auch mit jeder anderen Fläche. Wie viele zusammenhängende Code-Teile spezifisch zwischen den grün eingefärbten Dateien und der selektierten, blau eingefärbten Datei kopiert wurden, wird durch die Zahlen in den grünen Flächen angezeigt. Diesem Bild kann man unmittelbar entnehmen, dass es große Dateien mit einem sehr hohen Grad an Redundanz gibt und dass im Falle der aktuell selektierten Datei, die Redundanz nicht lokal ist, sondern gemeinsame Code-Teile über weit entfernte Dateien existieren.

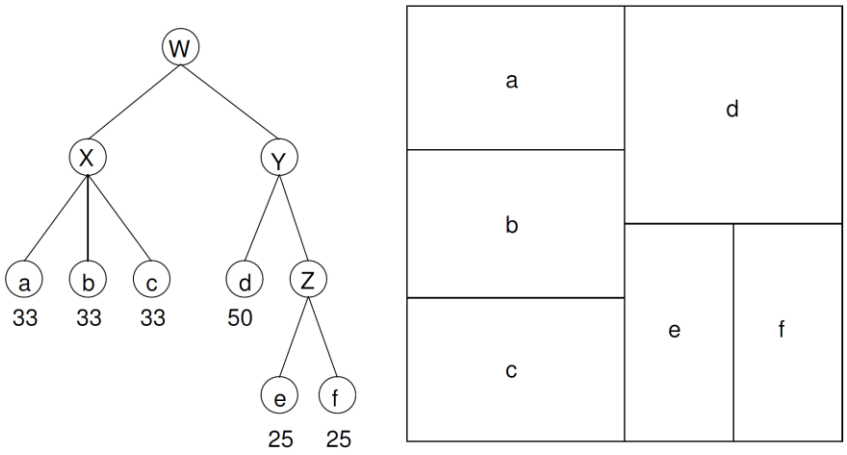

Abbildung 7: Beispiel einer Tree-Map

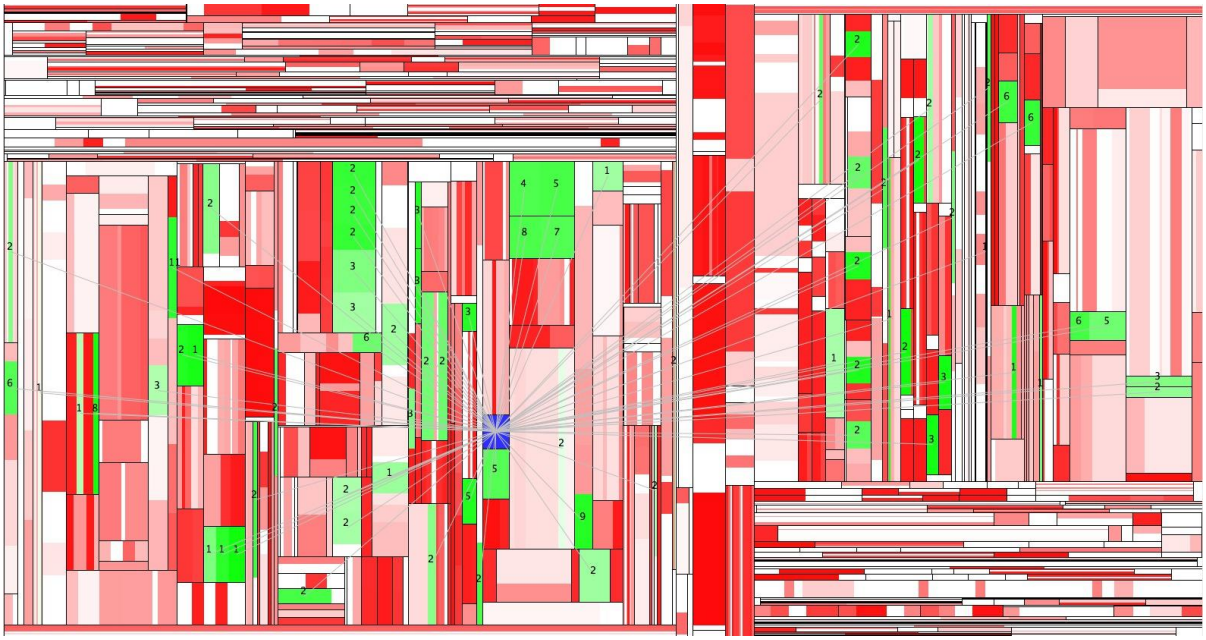

Abbildung 8: Reales Beispiel einer Tree-Map, die Copy&Paste anzeigt.

Der Vorteil der Tree-Maps ist es, dass sie sowohl proportional Metriken als auch zeitgleich die Hierarchie darstellen. Die Voraussetzung ist, dass die Metriken von den Blättern zu den inneren Knoten als Summe aggregiert werden können. Der gesamte zur Verfügung gestellte Raum wird effizient genutzt. Der Raum selbst kann zu Anfang vorgegeben werden. Es kann also nicht passieren, dass Informationen aus dem sichtbaren Bereich wandern. Die Skalierung ist nur begrenzt durch die Auflösung der Darstellung und der Fähigkeit des Betrachters, noch Punkte auseinanderhalten zu können. Selbstverständlich kann diese Darstellung auch leicht das Hinein- und Herauszoomen unterstützen.

Jedoch muss man sich mit dieser Visualisierung einige Zeit auseinandergesetzt haben, bis man sie leicht in allen Teilen verstehen kann. Gerade die Abgrenzung der Hierarchie in tief verschachtelten
Bäumen wird schnell schwierig. Außerdem sind die Metriken auf die Blätter begrenzt.

Tree-Maps hatten wir als eine erste Möglichkeit kennen gelernt, die Visualisierung von Graphen und Metriken zu kombinieren. Allerdings konnten damit keine allgemeine Graphen, sondern nur Bäume dargestellt werden. Binäre Relationen können allenfalls durch eingeblendete Kanten dargestellt werden, wie es das Beispiel in Abbildung 8 zeigte. Dort werden die Kanten aber nur für die aktuell selektierte Datei eingeblendet. Würde man alle Kanten gleichzeitig darstellen, wären die Knoten kaum mehr zu sehen.

Eine allgemeinere Visualisierung von Metriken und Relationen zugleich sind **polymetrische Sichten**. Polymetrische Sichten stellen binäre Relationen in Form von Graphen dar. Graphen sind in der Mathematik lediglich Knoten und Kanten. Sobald sie aber gezeichnet werden (vergleiche Abbildung 9), erhalten sie eine Darstellung, das heißt, ein Ausdehnung, Form und Farbe sowie einen Ort, an dem sie gezeichnet werden.

Diese visuellen Eigenschaften können verwendet werden, um weitere Informationen abzubilden.

Knoten haben in der zweiten Dimension eine Höhe und Breite sowie eine X- und Y-Position. Sie haben darüber hinaus eine Farbe. Mit diesen Mitteln können bis zu fünf Metriken dargestellt werden. Darüber hinaus könnte man noch Form und Textur für nominale Information und Farbverlauf für ordinale Information ausnutzen.

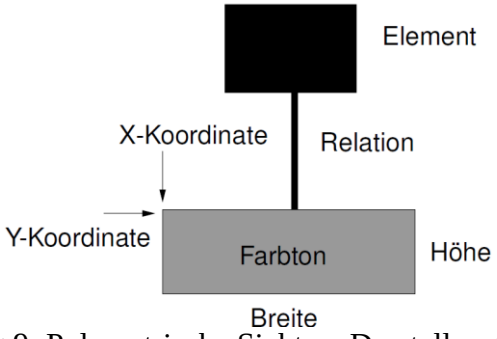

Abbildung 9: Polymetrische Sichten: Darstellungselemente

Ein Beispiel für eine polymetrische Sicht ist in Abbildung 10 zu sehen. Wir sehen hier Klassen als Knoten dargestellt. Die Vererbung ist durch Kanten von oben nach unten ausgedrückt. Die Breite der Knoten ist die Anzahl der Attribute einer Klasse und die Höhe die Anzahl der Methoden (jeweils exklusive der ererbten). Der Farbton bildet die Anzahl der Code-Zeilen der Klasse ab. Je dunkler die Klasse ist, desto mehr Code-Zeilen hat sie. Die schwarzen Pfeile markieren besonders interessante Klassen, die allein durch die Betrachtung als statistische Ausreißer herausstechen. Die mit 1 bezeichnete Klasse fällt durch ihre Höhe und dunkle Farbe sofort ins Auge. Offenbar handelt es sich um eine Klasse mit sehr vielen Methoden und einer Anzahl Code-Zeilen, wie sie sonst keine Klasse hat. Diese Klasse sollte untersucht werden, ob sie nicht verkleinert werden kann, indem sie in mehrere kleinere Klassen aufgeteilt wird. Die zweite Klasse ist im Gegensatz dazu sehr klein in Bezug auf alle drei Größenmetriken. Von ihr leitet auch keine andere Klasse ab. Es wäre hier zu untersuchen, ob diese Klasse überhaupt benötigt wird. Die dritte Klasse ist hoch und breit und hat auch im Verhältnis viele Zeilen Code. Es leitet keine Klasse davon ab. Möglicherweise könnte sie in mehrere Klassen, die voneinander erben, zerlegt werden. Bei der Klassenhierarchie, auf die der Pfeil mit der Nummer vier verweist, fällt auf, dass von zwei Klassen der zweiten Ebene sehr viele weitere Klassen ableiten, während die anderen Klassen dieser Ebene nicht weiter spezialisiert werden. Die fünfte Klasse ist winzig, hat aber zwei recht große Geschwisterklassen und sehr viele Kinderklassen. Das könnte ein Hinweis darauf sein, dass es sich hier um eine abstrakte Klasse handelt.

Das Beispiel zeigt, dass man mit der polymetrischen Sicht sehr viele unterschiedliche Metriken gleichzeitig darstellen und visuell vergleichen kann. Hier wird die menschliche Fähigkeit der visuellen
Mustererkennung im Sinne der Visual Analytics ausgenutzt. Durch beliebige Abbildung von Metriken auf die fünf visuellen Attribute Höhe, Breite, X- und Y-Koordinate sowie Farbe kann man sich seine eigene Sicht schaffen, die für den jeweiligen Zweck am besten geeignet ist.

Die Sicht ist jedoch auf fünf Metriken begrenzt und nicht in allen Fällen können die Koordinaten für Metrikwerte verwendet werden. Sobald Kanten gezeichnet werden, bevorzugt man eher ein Layout
der Knoten, das den Graph lesbar macht. Dann können die Koordinaten keine andere Semantik haben.

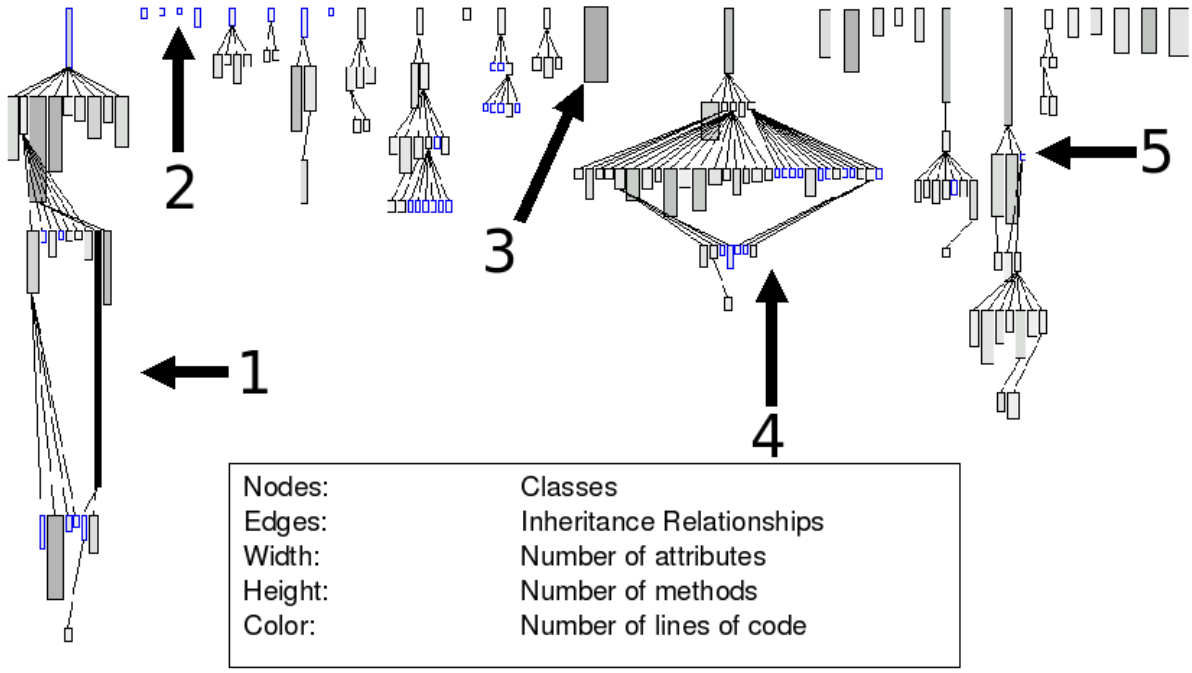

Abbildung 10: System Complexity View

Eine besondere Herausforderung bei Verwendung von Graphen ist die Visualisierung von Relationen. Will man zum Beispiel einen Aufrufgraphen visualisieren, dann liegt es nahe, dies als einen Graphen darzustellen, dessen Knoten die Funktionen und dessen Kanten die Aufrufe zwischen Funktionen repräsentieren. Wenn ein solcher Graph von einem Analysewerkzeug extrahiert wurde, dann haben die Knoten und Kanten jedoch zunächst keine natürlichen Positionen. Die Wissenschaft des Graphzeichnens (engl. *Graph Drawing*) beschäftigt sich mit Algorithmen, die Knoten und Kanten möglichst effizient so platzieren, dass daraus ein übersichtlicher Graph wird. Ein Beispiel für einen visualisierten Graphen sieht man in Abbildung 11. Der Algorithmus, der die Koordinaten der Knoten berechnet hat, betrachtet die Knoten als Körper und die Kanten als Anziehungskräfte, um dann iterativ eine Anordnung zu bestimmen, die ein Kräftegleichgewicht darstellt. Die Probleme hierbei sind, dass Hierarchien verloren gehen, die Anordnung nicht notwendigerweise die Semantik adäquat widerspiegelt und es zu Kreuzungen von Knoten und Kanten kommt. Es ist der Darstellung nicht ohne weiteres zu entnehmen, wer mit wem verbunden ist.

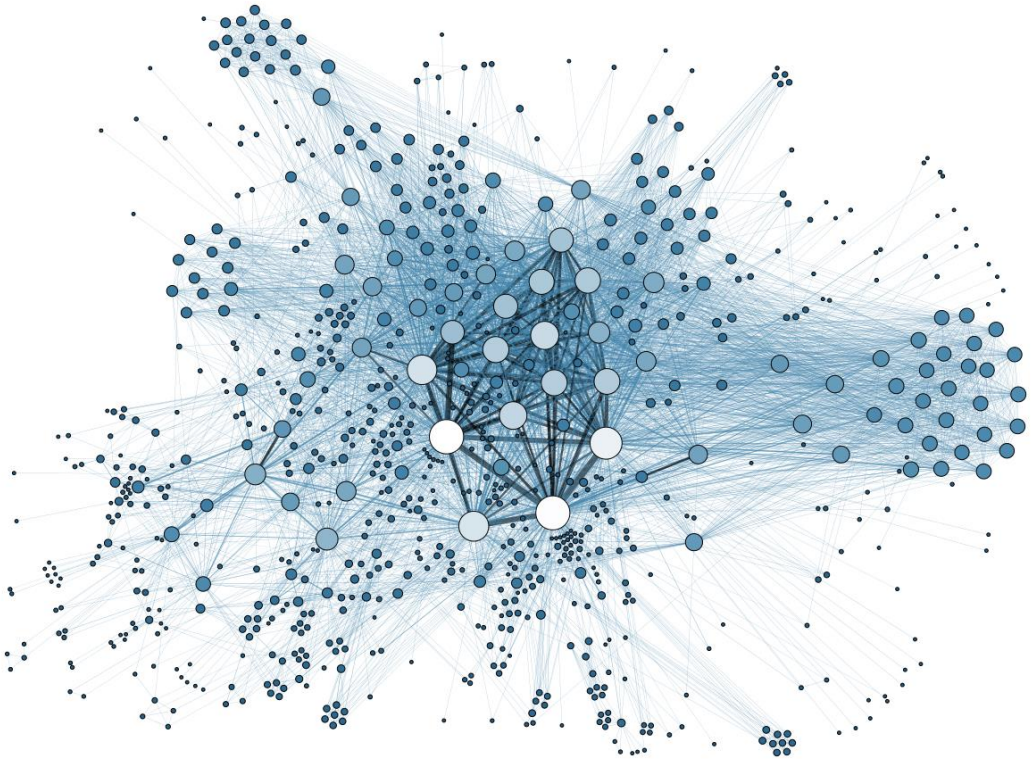

Abbildung 11: Ein automatisch angeordneter Graph (Quelle: DOI:10.3166/LCN.10.3.37-54)

Eine neue, recht vielversprechende Idee, die Hierarchie der Software zu erhalten und für mehr Übersichtlichkeit bei der Darstellung der Kanten zu sorgen, sind gebündelte Kanten. Ein Beispiel hierfür zeigt Abbildung 12. Die Hierarchie ist als geschachtelte Kreissegmente in Form äußerer Ringe wiedergegeben. In der Mitte finden sich die am tiefsten geschachtelten Elemente, wie zum Beispiel Klassen oder Dateien. In der Mitte ist dann Raum für die Kanten. Diese Anordnung stellt somit sicher, dass Kanten keine Knoten kreuzen können. Darüber hinaus werden die Kanten, die von nah beieinander liegenden Knoten ausgehen und zu nah beieinander liegenden Knoten hinführen, gebündelt. Der Grad der Bündelung kann vom Nutzer eingestellt werden. Im linken Teil von Abbildung 12 findet keine Bündelung der Kanten statt. Im rechten Teil wird gebündelt, wodurch die Darstellung deutlich aufgeräumter wirkt und auch schneller zu erkennen ist, welche Teile mit welchen anderen Teilen verbunden sind. Auch hier kann man jedoch nicht immer ohne weiteres erkennen, welche Knoten mit den Kanten verbunden werden. Dies wird erst durch Selektion von Knoten oder Kanten und dem Ausblenden nicht selektierter Elemente ermöglicht.

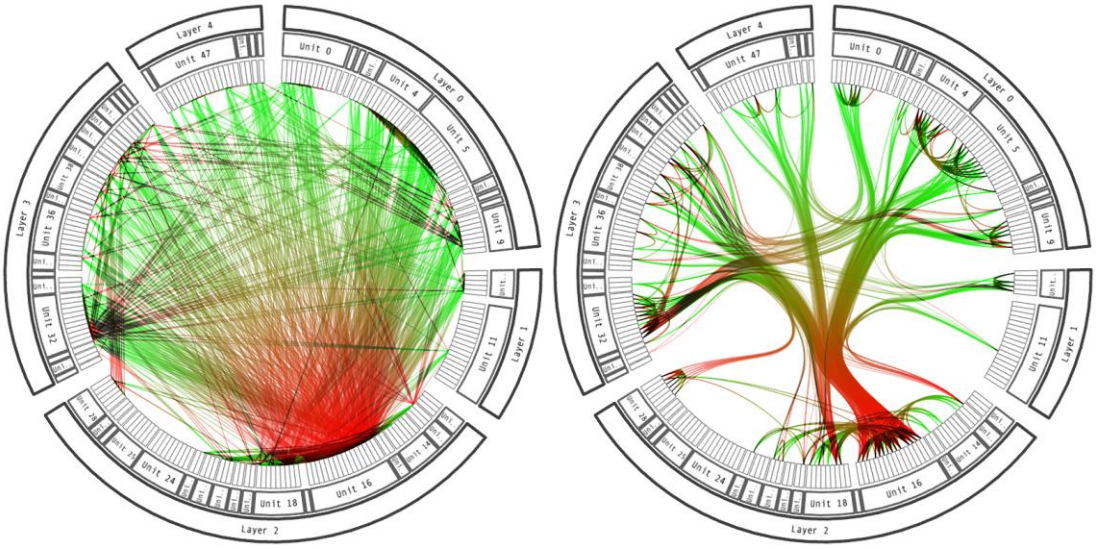

Abbildung 12: Gebündelte Kanten (Quelle: http://www.win.tue.nl/vis1/home/dholten/)

Welche Rolle spielen Metaphern?

Metaphern nutzen das Prinzip der Analogie zweier Gegenstandsbereiche und erlauben es, bekannte Aspekte des einen Bereichs auf den anderen Bereich zu übertragen. In der Mensch-Maschine-Kommunikation sind sie allgegenwärtig. Man denke nur an den Papierkorb auf dem Desktop. Beides – Papierkorb und Desktop (Schreibtisch) – sind Alltagsgegenstände, die in graphischen Benutzungsoberflächen nachgebildet sind. Metaphern werden auch in der Software-Visualisierung gerne eingesetzt. Eine der populärsten Metaphern sind die Software-Städte. Abbildung 13 zeigt ein Beispiel einer solchen Software-Stadt. Dabei werden Klassen eines objektorientierten Programms als Gebäude dargestellt. Wie bei den polymetrischen Sichten werden ausgewählte Metriken (z. B. Anzahl von Methoden, Attribut und Code-Zeilen) dargestellt, die auf die Breite, Länge und in diesem Fall durch die dritte Dimension hinzugekommene Höhe abgebildet werden. Letzten Endes sind Software-Städte nichts weiter als dreidimensionale Tree-Maps. Sie erwecken beim menschlichen Betrachter aber sofort den Eindruck einer Stadt amerikanischen Zuschnitts und wirken weit weniger abstrakt als zweidimensionale Tree-Maps.

Software-Cities eignen sich gut für die Darstellung von Aspekten einer bestimmten Version einer Software. Soll hingegen die Evolution einer Software über verschiedene Versionen hinweg dargestellt werden, sind sie weniger geeignet. Im Verlauf der Evolution können weitere Komponenten hinzukommen oder existierende Komponente wegfallen oder sich ihre Eigenschaften, die durch Höhe, Breite und Länge abgebildet sind, ändern. Tree-Maps sind für eine kompakte Repräsentation auf engem Raum ausgelegt. Das Layout sollte sich aber über verschiedene Visualisierungen von Versionen derselben Software möglichst nicht ändern, da sich ansonsten der menschliche Betrachter nicht mehr zurecht findet. Eine bessere Darstellung für diesen Zweck sind die Evo-Streets, wie sie in Abbildung 14 abgebildet sind. Die Gebäude sind hier zum Beispiel auch wieder Klassen eines Programms. Die sie verbindenden Straßen bilden den Hierarchiebaum ab. Je schmaler eine Straße ist, desto tiefer ist die dadurch dargestellte Hierarchieebene. Hier wird die Einschränkung aufgegeben, mit einer

vorgegebenen Fläche auszukommen. Stattdessen können am Rande Dinge hinzukommen. Die Evo-Streets können sich somit ausdehnen. Das Alter einer Klasse wird darüber hinaus noch durch einen Höhenzug dargestellt. Im Verlaufe der Evolution wächst die Stadt auf einem Gebirge.

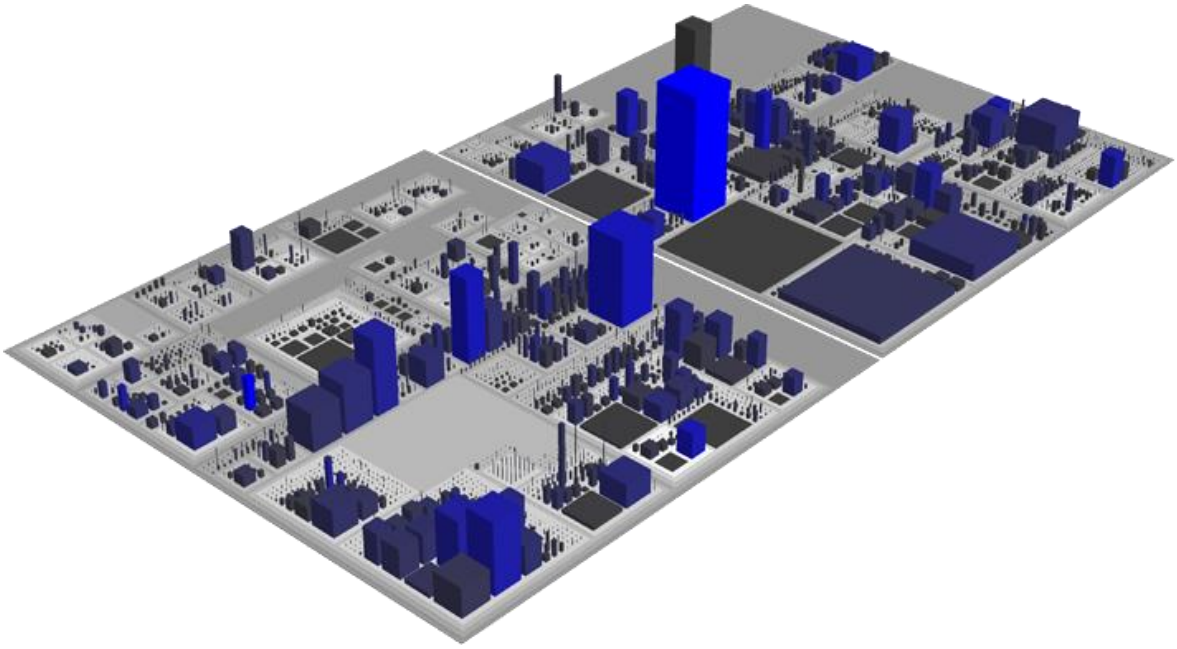

Abbildung 13: Software-Stadt (Quelle: http://wettel.github.io/codecity.html)

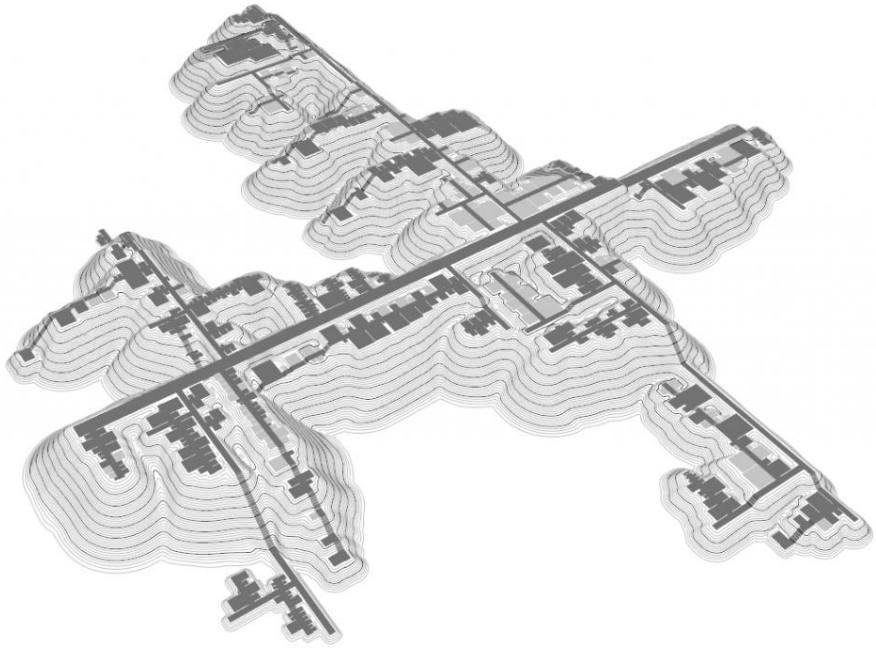

Abbildung 14: (Quelle: http://software-cities.org/)

Wie kann ich meine eigene Software-Visualisierung selbst implementieren?
Es gibt eine Reihe von kostenlosen und kostenpflichtigen Programme und Frameworks, mit denen man seine eigenen Visualisierungen gestalten kann. Eine einfache Suchanfrage im Internet mit den Schlüsselwörtern „visualization tools“ liefert gleich viele Treffer. Wenn es um Visualisierung in Kombination mit analytischen Methoden geht, dann ist eines der populärsten Frameworks R (https://www.r-project.org/). R ist ein Open-Source-basiertes Ökosystem von Programmen für statistische Verfahren und Visualisierung. Wer Visualisierungen gestalten möchte, die interaktiv im Browser laufen, wird in der Regel bei D3.js fündig werden (https://d3js.org/). Die genannten Frameworks helfen sehr bei der Visualisierung. Jedoch muss man darüber hinaus auch die Daten extrahieren, die visualisiert werden sollen. Zudem muss die Visualisierung auch zur Problemstellung passen. Das Design der Visualisierung selbst und die Extraktion der Daten nehmen einem die Visualisierungs-Tools nicht ab.

Zusammenfassung
Software-Visualisierung ist in der Wartung unabdingbar, wenn es um Probleme gibt, bei denen ein Mensch eine sehr große Menge von Daten interpretieren muss. Eine spezifische Software-Visualisierung ist immer nur gut für bestimmte Zwecke und schlecht für andere, weil jede Visualisierung bestimmte Information betont und andere Information vernachlässigt. Wann sich eine Visualisierung (oder auch Notation) eignet, besagt die Match-Mismatch-Hypothese von Gilmore und Green:

> *“Problem-solving performance depends on whether the structure of a problem is matched by the structure of a notation.”*

Man sollte also viele mögliche Varianten von Visualisierungen kennen, so dass man die dem Problem angemessene ausfindig machen kann. So wenig wir erwarten würden, dass ein Laie in der Lage ist, Details in einem Ultraschallbild zu erkennen, so wenig sollten wir wohl auch vermuten, dass ein Wartungsprogrammierer beim ersten Blick auf einen speziellen Typ von Software-Visualisierung im Stande ist, etwas zu erkennen. Eine Visualisierung zu verstehen, ist Übungssache. Man spricht hier auch von einem Alphabetismus der Software-Visualisierung. Wir müssen lernen, eine graphische Sprache zu verstehen und uns darin auszudrücken. Und was auf den ersten Blick bunt und schön aussieht, kann sich umgekehrt bei der näheren Beschäftigung als unnütz herausstellen. Schließlich sei noch angemerkt, dass wir Menschen noch mit vielen weiteren Sinnen ausgestattet sind, von denen wir bei der Darstellung von Software aber kaum Gebrauch machen. Möglicherweise werden wir in der Zukunft unsere Software sogar riechen, schmecken und fühlen können. Tatsächlich gibt es schon erste Ansätze, Software auch zu hören. Zudem ist zu erwarten, dass bald Visualisierungen in Virtual Reality aufkommen werden. Die notwendigen Geräte dafür sind bereits erschwinglich und einige Forschergruppen – unter anderem auch unsere – experimentieren bereits damit.

Weiterführende Literatur
Weiterführende Literatur findet man bei Caserta und Zendra (2011), die eine Literaturübersicht zur Visualisierung von statischen Aspekten von Software verfasst haben. Ghanam und Carpendale (2008) beschreiben Visualisierungen speziell von

Software-Architektur. Eine Übersicht zum Einsatz dreidimensionaler Darstellungen in der Software-Visualisierung geben Teyseyre und Campo (2009). Ein umfassendes Buch zum Thema Software-Visualisierung wurde von Diehl (2007) veröffentlicht.

Pierre Caserta, Olivier Zendra. Visualization of the Static aspects of Software: a survey. In: IEEE Transactions on Visualization and Computer Graphics, Institute of Electrical and Electronics Engineers, 2011, 17 (7), Seiten 913-933.

Alfredo R. Teyseyre and Marcelo R. Campo. An Overview of 3D Software Visualization. In: IEEE Transactions on Visualization and Computer Graphics, 2009, 15(1), Seiten 87-105.

Yaser Ghanam and Sheelagh Carpendale. A Survey Paper on Software Architecture Visualization. Technical report, University of Calgary, 2008. https://dspace.ucalgary.ca/bitstream/1880/46648/1/2008-906-19.pdf.

Stephan Diehl. Software Visualization. 2007. Springer Verlag.

Autor

Prof. Dr. Rainer Koschke ist Leiter der Arbeitsgruppe Softwaretechnik an der Universität Bremen. Sein Forschungsschwerpunkt ist die Wartung und Evolution existierender Software. Praktisch einsetzbare Methoden und Werkzeuge für die Weiterentwicklung existierender Software sind das Ziel seiner Forschung. Er ist Mitgründer der Axivion GmbH. Axivion (http://www.axivion.com) bietet Lösungen zur Sicherung der inneren Software-Qualität und der Software-Wartbarkeit an. Im Zentrum steht dabei die Axivion Bauhaus Suite, mit der Software-Erosion effizient und effektiv bekämpft wird.