Hochschule für Angewandte Wissenschaften Hamburg
Hamburg University of Applied Sciences

Grundseminar

Edgar Toll

Software Architecture Discovery

Fakultät Technik und Informatik
Studiendepartment Informatik

Faculty of Engineering and Computer Science
Department of Computer Science

Edgar Toll

Software Architecture Discovery

Grundseminar eingereicht im Rahmen der Masterprüfung

im Studiengang Master of Science Informatik
am Department Informatik
der Fakultät Technik und Informatik
der Hochschule für Angewandte Wissenschaften Hamburg

Betreuender Prüfer: Prof. Dr. Tim Tiedemann
Zweitgutachter: Prof. Dr. Kai von Luck

Eingereicht am: 09. Mai 2017

Edgar Toll

Thema der Arbeit
Software Architecture Discovery

Stichworte
Software, Architektur, Architektur Wiederentdeckung, Software Engineering

Kurzzusammenfassung
In einer Zeit, in der immer mehr Software Projekte älter und komplexer werden, steigt auch der Bedarf, diese weiterhin verstehen zu können. Auch werden Projekte nicht mehr gewartet oder betreut. Teile dieser Softwareprojekte können jedoch noch von Nutzen sein oder wiederverwendet werden. Dazu muss die Software jedoch verstanden werden können. Dazu werden in dieser Arbeit genutzte Ansätze zur Informationsgewinnung aus Softwareprojekten betrachtet.
Edgar Toll

Title of the paper
Software Architecture Discovery

Keywords
Software, Architecture, Architecture Discovery, Software Engineering

Abstract
In a time where more and more software projects are getting older and more complex the need for understanding them is getting higher. Also Projects are abandoned or aren't under active development anymore. But parts of the software projects are of use for other parties. In order to obtain these parts, the software has to be understood. Currently used approaches in order to obtain these information needed from software projects are looked at in this paper.

Inhaltsverzeichnis

Listings

1 Einführung

Ein Softwareprojekt wandelt sich über die Zeit. Kunden haben neue Anforderungen, die nicht von Beginn an eingeplant werden konnten oder neue Technologien werden in ein Projekt integriert. Da Softwareprojekte eine Komplexität haben, die ein einzelner Entwickler nicht mehr überblicken kann, wird es bei dem Integrieren von neuen Features immer zu Unstimmigkeiten kommen. Diese Unstimmigkeiten werden teilweise durch Software Tests aufgefangen und können behoben werden, sind teilweise aber auch in der Architektur, welche nicht von Tests abgesichert wird.

Derartige Unstimmigkeiten in der Architektur sind im Quellcode so nicht erkennbar, da sie abstrakter darüber liegt. Daher muss die bestehende Software auf die bestehende Architektur analysiert werden und mit der Soll-Architektur verglichen werden.

Dafür stellen sich 3 Fragen:

- Was will man wissen?
- Wie kommt man an diese Information?
- Wie stellt man die Ergebnisse dar?

Was will man wissen? Generell sind alle Qualitätsmerkmale von Software von Interesse. Beispielsweise können einen große Klassen oder Methoden interessieren, die schwer wiederverwendbar sind. Die Qualitätsmerkmale von Software sind bereits seit Jahren relativ unverändert dokumentiert ISO (1991) und werden nicht Teil dieser Ausarbeitung sein.

Wie stellt man die Ergebnisse dar? Um für Entwickler nutzbringend zu sein, müssen die Ergebnisse hilfreich aufbereitet werden. Die Darstellung von Metriken und dessen Verbesserung der Verständlichkeit für den Menschen wird beispielsweise von Psychologen oder in der Data Science untersucht. In Lilienthal (2008) wird das Thema angeschnitten und erläutert, dass Modularität, Mustertreue und Geordnetheit eine wichtige Rolle für das Verständnis für den Menschen spielen. Dieses Teilgebiet wird ebenfalls nicht in dieser Ausarbeitung betrachtet.

Wie kommt man an diese Informationen? Da die Architektur oder die Qualität der Software nicht im Quellcode zu sehen ist, muss diese Information extrahiert werden. In den folgenden Kapitel werden vier grundlegende Arten von Ansätzen erläutert.

Im vorletzten Kapitel Kapitel 3 „Relevante Konferenzen“ wird auf relevante Konferenzen zu diesem Thema eingegangen und im Abschlusskapitel Kapitel 4 „Fazit“ ein Fazit zum aktuellen Stand der Thematik gegeben.

2 Software Analyse Ansätze

Im folgenden Kapitel werden die vier meist genutzten Ansätze zur Software Analyse erläutert. Dabei es sich um die statische Analyse von Quellcode, der dynamischen Analyse von Software zur Laufzeit, der hybriden Analyse, also der gemeinsamen Nutzung von statischer und dynamischer Analyse und der vergleichsweise neuen Herangehensweise mit Graphdatenbanken.

Zu den jeweiligen Ansätzen werden Tools und/oder hervorzuhebende Projekte verwiesen, als auch deren Grenzen aufgezeigt.

2.1 Statische Quellcode Analyse

In der statischen Quellcode Analyse wird der Quellcode eingelesen und analysiert. Beispielsweise das Analysieren, wie viele Lines of Code eine Datei hat, ist bereits eine statische Quellcode Analyse. Auch nutzen viele Compiler eine statische Analyse um zum Beispiel auf ungenutzte Methoden hinweisen zu können.

Komplexere Tools verwenden Metasprachen um die zu Grunde liegende Programmiersprache zu abstrahieren. Solche Tools können mehrere, unterschiedliche Sprachen in Metasprachen umwandeln um so ein breiteres Feld an Programmiersprachen abdecken zu können. Jedoch verwenden die Hersteller der Analysetools jeweils eigene Metasprachen und zum Zeitpunkt des Schreibens ist kein Standard definiert oder in Aussicht.

Die Metasprachen sind jedoch meist sehr begrenzt und nah an den Programmierkonzepten, sodass sich diese schwer auf andere Programmierkonzepte anwenden lassen. So unterstützen Metasprachen für Java meistens auch .NET, seltener aber C++ oder ähnliche hardwarenähere Sprachen.

2.1.1 Code City

Besonders hervorstechend unter den statischen Analysetools ist *Codecity* Wettel und Lanza (2008). *Codecity* verfolgt den Ansatz, den Quellcode stadtähnlich darzustellen, wohingegen andere statische Analysetools eher auf unvisualisierte Metriken setzen. Die Stadtdarstellung ist in Abbildung 2.1 am Beispiel der JDK zu sehen. Dabei stellen die unteren, grauen Lagen

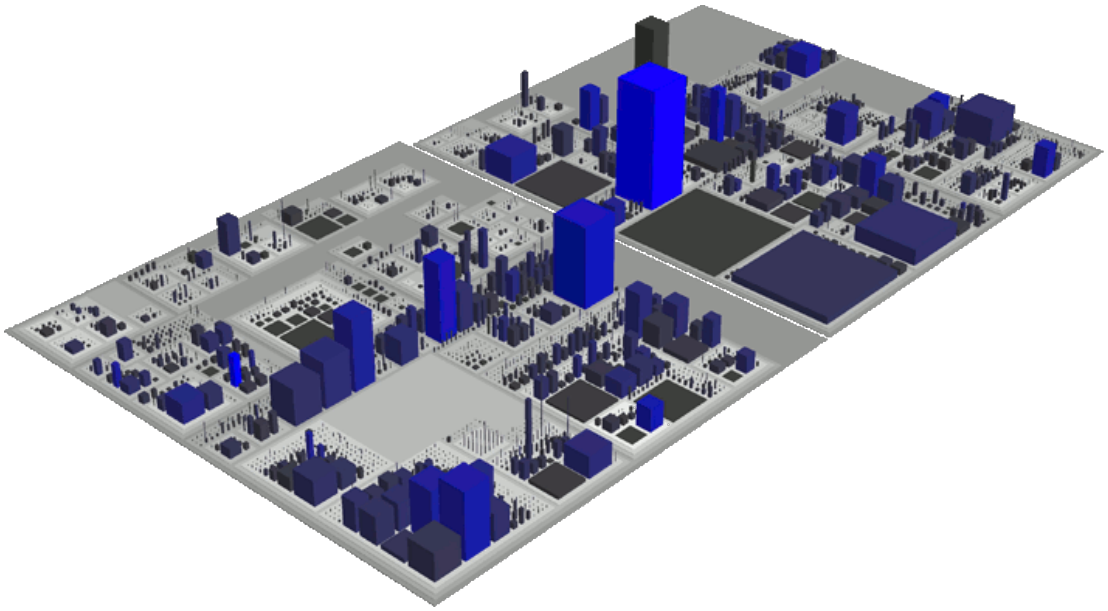

Abbildung 2.1: *Codecity* Darstellung des JDK Wettel und Lanza (2008)

die Namespaces dar, in denen die Klassen sich befinden. Die blauen „Gebäude“ stellen die Klassen dar. Ihre Höhe und Grundfläche wird dabei vom Umfang einer Klasse bestimmt. Viele Konstanten führen dabei zu einer großen Grundfläche und die Höhe wird durch die Anzahl der Methoden bestimmt. Die Lines of Code der Klasse resultieren in unterschiedlichen Blautönungen.

Codecity veranschaulicht so ein Softwareprojekt auf eine andere Art und Weise, als man es durch den Blick auf den direkten Quellcode sieht. So können schnell beispielsweise besonders komplexe Klassen gefunden werden und ein Refactoring zur Aufteilung in jeweils einfachere Unterklassen begonnen werden.

```
1 public class Human {
2   public Pet[] Pets;
3 }
4
5 public class Pet {
6   public Human getOwner();
7 }
```

Listing 2.1: Code Beispiel eines Aufruf Zyklus

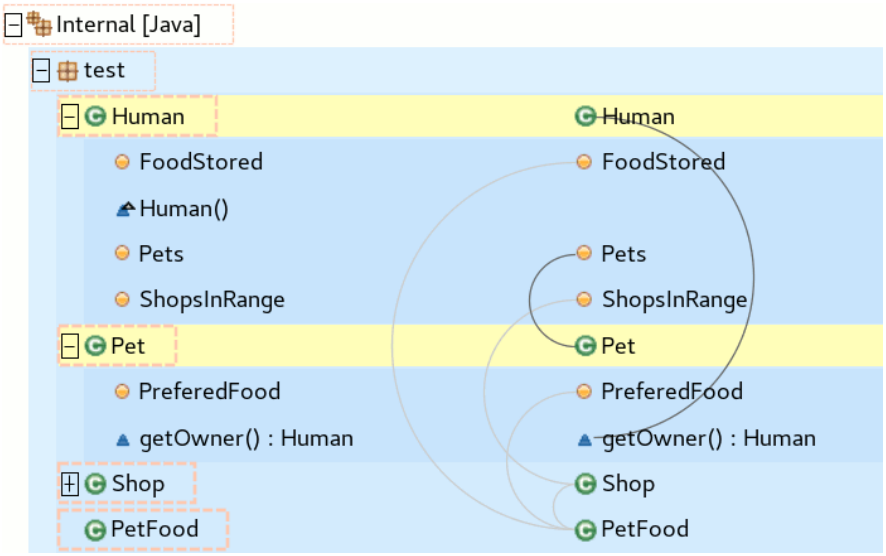

Abbildung 2.2: *Sonargraph* am eigenen Beispiel

2.1.2 Sonargraph

Eine sonst auch häufig anzutreffende statische Quellcode Analyse ist das bilden von hierarchischen Aufrufstrukturen. Methodenaufrufe aus einer Klasse oder eines Packages in eine andere Klasse oder Package werden dabei analysiert, um Strukturen zu erkennen. So lassen sich im Idealfall Abstraktionsschichten erkennen, da eine unten liegende Schicht keine höhere Schicht aufrufen sollte. Ist dies bei der Entwicklung jedoch trotzdem (unbemerkt) passiert, können diese Fehler so entdeckt und behoben werden.

In Listing 2.1 ist beispielhaft ein Aufruf Zyklus implementiert. Ein Austausch einer der beiden Klassen bedingt dabei immer auch den Austausch oder Anpassung der jeweils anderen Klasse. Dies ist jedoch vor allem in größeren Projekten nur schwer im Quellcode erkennbar. Tools wie der *Sonargraph* [hello2morrow.com/products/sonargraph] können diese Zyklen jedoch sehr einfach aufzeigen.

Das in Listing 2.1 verwendete Beispielprojekt wurde in Abbildung 2.2 mit dem *Sonargraph* importiert und dargestellt. Dabei werden von höher liegenden Klassen nach unten in den linken Bögen dargestellt. Aufrufe von unten liegenden Klassen nach oben werden mit den rechten Bögen dargestellt. So ist hier klar der Verstoß gegen zyklenfreien Quellcode erkennbar.

Ähnliche Analysetools wie der *Sonargraph*, so zum Beispiel *Lattix* (Abbildung 2.3) oder *Structure101* (Abbildung 2.4), arbeiten mit unterschiedlichen Metasprachen, verfolgen jedoch nahezu die gleichen Ziele. Beispielsweise hat die Matrixdarstellung von *Lattix* die gleiche Aussage wie die Bogendarstellung des *Sonargraph*.

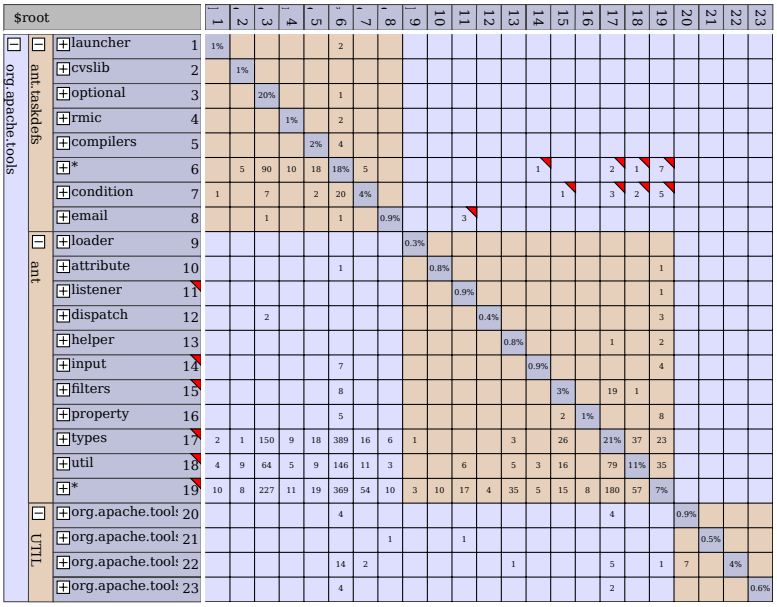

Abbildung 2.3: Betrachtung einer Software in einer Matrix Darstellung mit *Lattix* [lattix.com]

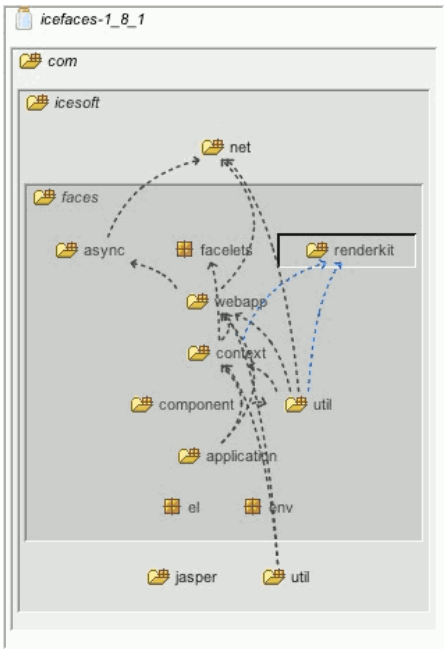

Abbildung 2.4: *Structure101* [structure101.com]

2.1.3 Fazit und Einschränkungen

Auffallend ist der hohe Anteil an Tools für Hochsprachen wie Java oder .NET. Tools für Hardware nähere Sprachen sind seltener vertreten. Möglicherweise lässt sich dies durch die komplexere Quellcodestruktur oder auch durch die häufiger verwendete Hochsprache als Programmiersprache erklären.

Die statische Quellcode Analyse beschränkt sich rein auf den Quellcode und kann daher keine Events von außen berücksichtigen. Ist ein Kontrollpfad theoretisch möglich, so wird dieser berücksichtigt, auch wenn er in der Realität möglicherweise nie auftritt. Dies kann jedoch von der dynamischen Analyse erkannt werden, die im folgenden Abschnitt beschrieben wird. Mit der Kombination aus statischer und dynamischer Analyse, der hybriden Analyse, beschrieben in Abschnitt 2.3, können so die Stärken beider Ansätze kombiniert werden.

2.2 Dynamische Analyse

In der dynamischen Analyse wird ein Programm ausgeführt und sein Verhalten analysiert. Ein bekannter Ansatz ist das Beobachten von Nachrichten zwischen den Komponenten eines Systems. Dieser Ansatz gewinnt zunehmend an Bedeutung, da aktuell in angestrebten Architekturen die Komponenten voneinander entkoppelt sein sollen, wie zum Beispiel bei der Verwendung von Mikroservices.

Eine anschauliche Visualisierung von dynamischer Analyse verwendet *gotrace* Danyliuk. Das Ziel von *gotrace* ist die Visualisierung von Nebenläufigkeit in der Programmiersprache Go.

Ein häufig genutztes Pattern, Producer Consumer Pattern, stellt *gotrace* beispielsweise wie in Abbildung 2.5 zu sehen dar. Dabei sind die senkrechten Linien die Go Routinen, generell auch als eine Komponente des Systems zu sehen. Die blauen Pfeile stellen die Nachrichten und ihre jeweilige Sequenznummer dar. In diesem Beispiel produzieren die beiden Producer jeweils Nachrichten, die gesammelt werden und vom Consumer angenommen werden.

Zwei weitere Bespiele für die Visualisierung von Architekturen sind eine Primzahlberechnung über Filter Komponenten in Abbildung 2.6 oder einer Client Server Architektur in Abbildung 2.7

So kann, ohne den Quellcode zu kennen, bereits eine Architektur aus den zur Laufzeit anfallenden Daten erkannt werden. Diese Architekturen sind jedoch meist komplexer als einfache Programmierpattern und vermischen eine Vielzahl dieser. Der Aufwand der automatischen Erkennung von Architektur Pattern ist dem Nutzen aus aktueller Sicht nicht gerechtfertigt und zum aktuellen Zeitpunkt Bedarf dieser Ansatz weiterhin menschliche Genialität.

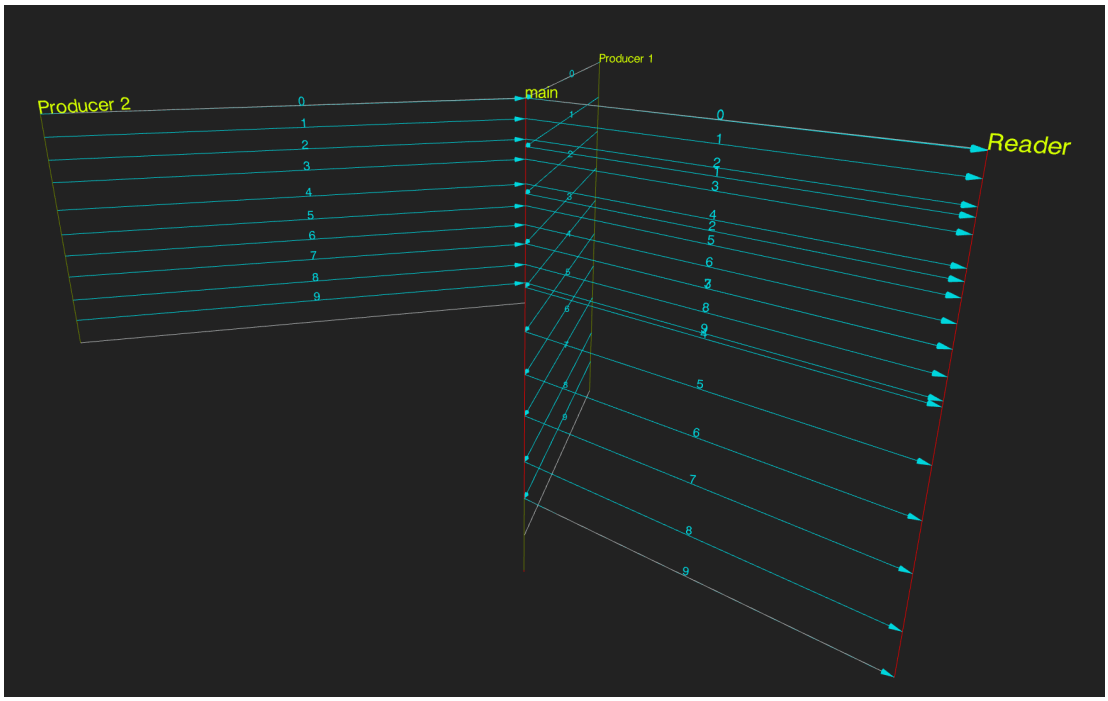

Abbildung 2.5: Betrachtung der Nachrichten mit dem Producer Consumer Pattern

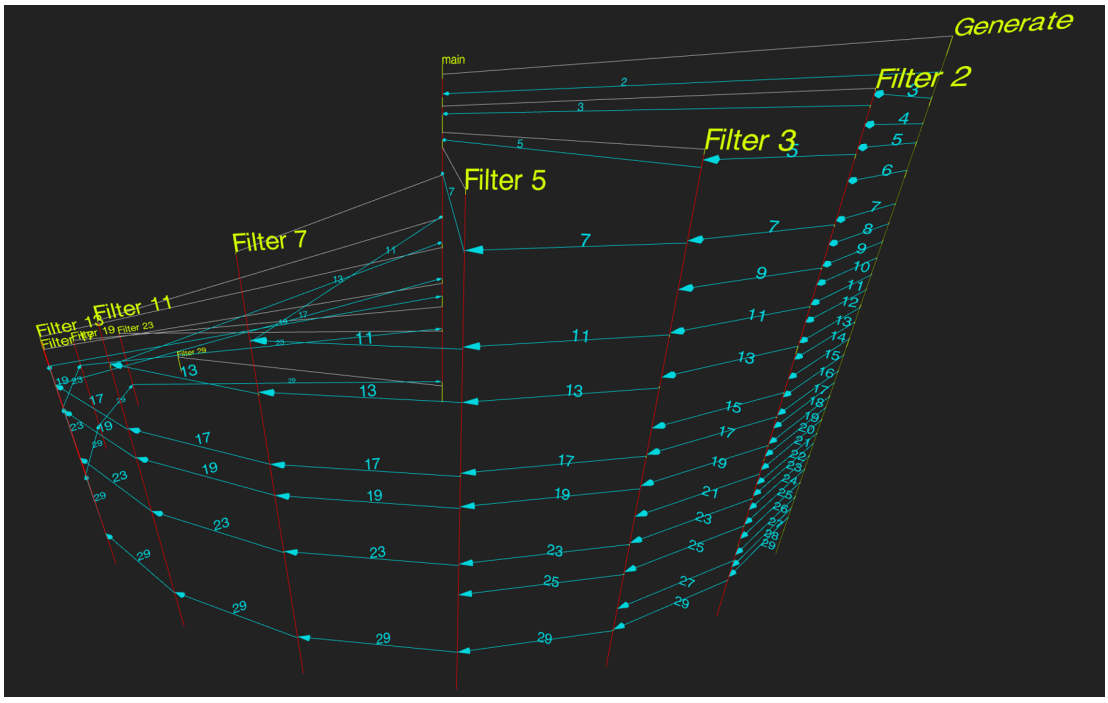

Abbildung 2.6: Betrachtung der Nachrichten einer nebenläufigen Primzahlberechnung

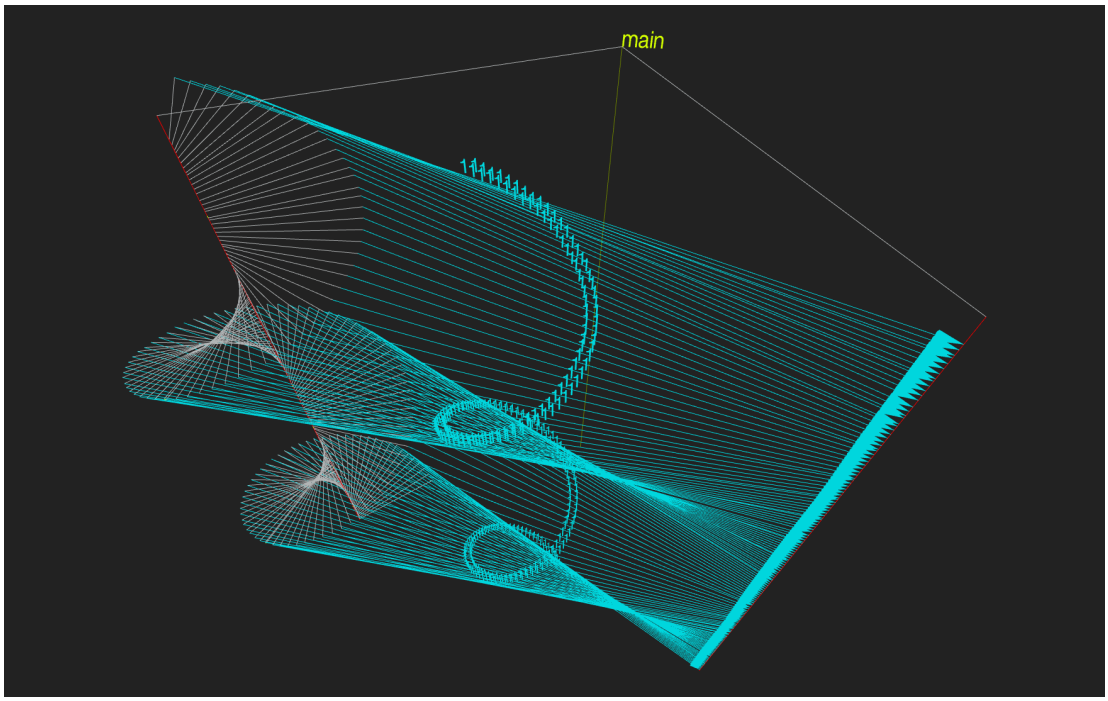

Abbildung 2.7: Betrachtung der Nachrichten von vielen Clients an einen Server

Ähnlich wie bei den statischen Quellcode Analysetools wurde beobachtet, dass der Anteil an Tools für Hochsprachen am Größten ist.

Da in der dynamischen Analyse die möglicherweise verfügbaren Daten des Quellcodes nicht genutzt werden, werden in der Praxis häufig beide Ansätze kombiniert eingesetzt. Dies wird als hybride Analyse bezeichnet und im folgenden Kapitel betrachtet.

2.3 Hybride Analyse

Im Kontrast zur statischen Quellcode Analysen werden bei der dynamischen Analyse keine bekannten Daten genutzt, sondern ausschließlich zur Laufzeit entstehende Daten beobachtet. Analysiert man eigene Software Projekte ist der Quellcode jedoch verfügbar und man nutzt so nur einen Teil des Potentials. Daher wird in der Praxis häufig eine Kombination aus beiden Ansätzen, die hybride Analyse, eingesetzt.

Ein bekanntes Beispiel für die hybride Analyse ist der Call Stack, den man zum Beispiel bei einer Exception in Java sieht. Der reale Verlauf der Aufrufe mit den Methodennamen aus dem Quellcode im Call Stack dargestellt, um dem Entwickler das finden der Fehlerquelle zu erleichtern.

Mit dem Kieker Framework van Hoorn u. a. (2012) haben hauptsächlich die Uni Kiel und die Uni Stuttgart den Beginn eines quelloffenen, hybriden Analysetools für Java geschaffen. Es verbindet die Vorzüge von statischer und dynamischer Analyse.

Alle anderen hybriden Analysesysteme, die zum Zeitpunkt der Recherche betrachtet wurden, sind nicht Open Source / zu Forschungszwecken bereitgestellt und daher nicht näher betrachtet worden.

2.4 Graphdatenbanken

Ähnlich wie die Verwendung von Graphdatenbanken in der Informatik ist der Ansatz zur Architekturanalyse verhältnismäßig jung.

```
1 int bar(int);
```

Listing 2.2: foo.h

```
1 #include "foo.h"
2 int bar(int input) {
3   return input;
4 }
```

Listing 2.3: foo.c

```
1 #include "foo.h"
2 int main(int argc, char **argv) {
3   return bar(argc);
4 }
```

Listing 2.4: main.c

Das Framework *Frappe* Hawes u. a. (2015) wird von Oracle zur Linux Kernel Analyse entwickelt. Dabei erkennt es Beziehungen zwischen Elementen im Quellcode und speichert diese in einem Graphen ab. So extrahiert *Frappe* den in Listing 2.4 gezeigten Quellcode in den in Abbildung 2.8 gezeigten Graphen. Die Beziehungen sind dabei unterschiedlicher Art. Zum einen werden Beziehungen des Build Vorganges gespeichert, wie der *foo.o* Datei und dessen Beziehungen. Andererseits werden auch Beziehungen aus dem statischen Quellcode gespeichert, wie der *main* Methode, dessen Parameter und den Aufruf an die *bar* Methode.

Der Ansatz von *joerg* Yamaguchi u. a. (2014) oder auch *Joerg Analysis Platform* genannt, ist verglichen mit *Frappe* sehr ähnlich. *joerg* wird von der Uni Göttingen entwickelt und dient zur Findung von Bugs in großen C/C++ Projekten, wie zum Beispiel einem Betriebssystem Kernel.

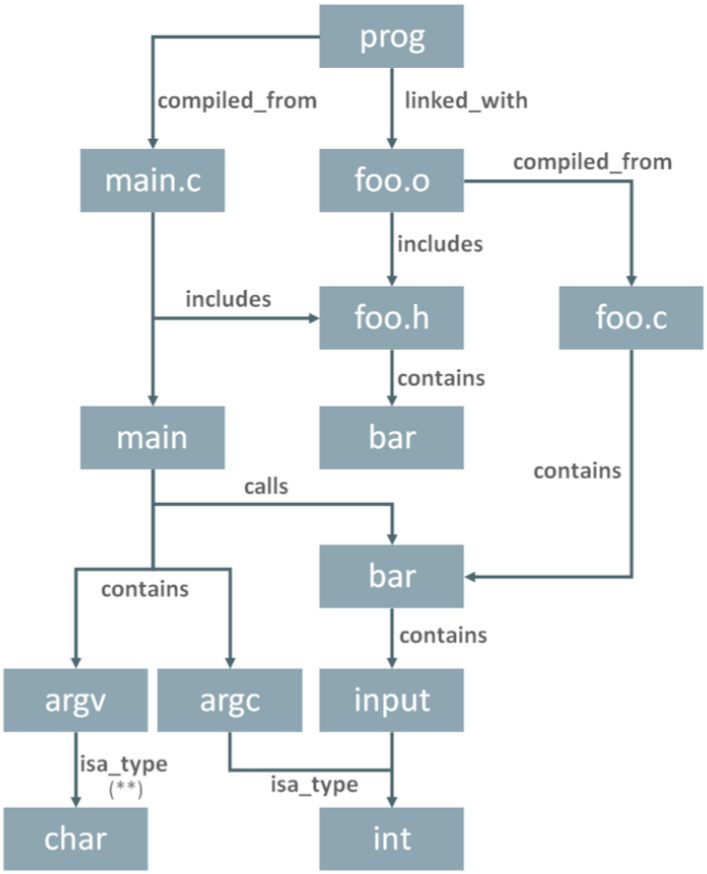

Abbildung 2.8: C-Code gespeichert in Graphdatenbank mit *Frappe*

Dafür wird, ähnlich wie bei *Frappe*, der Quellcode in eine Graphdatenbank geladen. Diese Graphdatenbank kann dann mit speziellen Abfragen aus bestimmte Beziehungen zwischen Knoten geprüft werden.

Beispielsweise kann eine Suchabfrage die Abfrage eines Array Indexes beinhalten, der zuvor nicht auf einen Overflow geprüft wurde. Daher funktioniert dieser Ansatz aktuell, ähnlich wie bereits bei der statischen Quellcode Analyse beschrieben, nur mit menschlicher Genialität. Bekannte Bugs können so als Suchanfrage gestellt werden und weitere, ähnliche Fälle des bekannten Fehlers aufdecken.

Eine interessante Beobachtung dabei ist, dass im kompletten Gegensatz zu statischer und dynamischer Analyse zum Zeitpunkt der Recherche nur Open Source Tools für C/C++ existieren. Statt übermäßig vielen Tools für Hochsprachen existieren hier nur Hardware nahe Analysetools.

3 Relevante Konferenzen

Es gibt wenig Konferenzen, die sich zentral mit Architekturen beschäftigen, speziell zur Software Architecture Discovery gibt es keine Konferenzen. Die meisten Konferenzen sind auf Programmiersprachen spezialisiert und haben unter anderem Vorträge zu Architekturthemen wie auch der Architecture Discovery.

Als Architektur Konferenz sind zum Beispiel die SATURN (Denver, 1-4 Mai 2017), die SANER (Klagenfurt / Österreich, 20-24 Februar 2017) oder die QCon (New York, 26-30 Juni 2017; London 5-9 Mai 2018) vertreten.

Für Programmiersprachen Konferenzen, auf der unter anderem Architekturvorträge laufen, gibt es wesentlich mehr Beispiele. Hervorzuheben ist hier die Java Konferenz JAX (Mainz, 8-12 Mai 2017), da dort die Firma hinter dem *Sonargraph* einen Vortrag über eben diesen hatten. Auffällig ist die Tatsache, dass die Architekturvorträge auf den Programmiersprachen Konferenzen meist sehr sprachnah und nicht allgemein gehalten sind.

Für die Wichtigkeit von Architekturen und Pattern in der Informatik gibt es vergleichsweise wenig generell gehaltene Konferenzen zu dem Thema. Gerade der Wandel in Richtung Container basierten Komponenten, die theoretisch jeweils unterschiedliche Programmiersprachen haben können und Mikroservices, die jeweils nur gemeinsame Schnittstellen benötigen, bedarf allgemein gehaltener Architekturdiskussionen.

Bisher nicht genutztes Protential steckt auch in der Komponenten- und Programmiersprachenübergreifenden Analyse mit Graphdatenbanken. Die Graphdatenbanken bieten ein gemeinsame Basis in die alle Komponenten / Programmiersprachen einzeln hinzugefügt und später verknüpft werden können. Dieses Potenzial wird jedoch zum Zeitpunkt der Recherche weder in einem wissenschaftlichen Paper noch einem Analyse Tool angesprochen oder verwendet.

4 Fazit

Zur Erkennung von Architekturen in Softwareprojekten exisitieren mehrere Ansätze. Jede dieser Ansätze benötigt jedoch in einem gewissen Maße menschliche Genialität um effektive Ergebnisse zu bekommen. Außerdem ist ein großer Teil der Tools kommerziell und nicht für die Forschung erstellt worden bzw. nicht Open Source. Stark erkennbar ist außerdem die deutlich höhere Anzahl an Tools für die viel genutzten Hochsprachen Java oder .NET gegenüber Hardware näheren Programmiersprachen.

Die Konferenzen zeigen sehr stark, das Architektur Pattern viel in Verbindung zu Programmiersprachen gesehen werden und vergleichsweise selten abstrakt und alleine stehend formuliert sind. Auch die Analysetools sind sehr stark an Programmiersprachenparadigmen gebunden. So funktionieren viele Tools entweder nur mit Hochsprachen wie Java oder .NET oder sind nur mit Hardware näheren Sprachen wie C/C++ verwendbar. Die Analyse mit Hilfe von Graphdatenbanken könnte programmiersprachenunabhängig genutzt werden, jedoch wird dies bisher nicht verfolgt.

Die Wichtigkeit der Software Architecture Analysis wird mit zunehmender Anzahl an Software Projekten in den kommenden Jahren und Jahrzehnten steigen. Aktuell scheint das Thema in der Praxis jedoch nur eine Niesche zwischen den Programmiersprachen darzustellen.

Literaturverzeichnis

[Danyliuk] Danyliuk, Ivan: Visualizing Concurrency in Go. URL https://divan.github.io/posts/go_concurrency_visualize/. – Forschungsbericht

[Hawes u. a. 2015] Hawes, Nathan ; Barham, Ben ; Cifuentes, Cristina: FrappÉ: Querying the Linux Kernel Dependency Graph. In: *Proceedings of the GRADES'15*. New York, NY, USA : ACM, 2015 (GRADES'15), S. 4:1–4:6. – URL http://doi.acm.org/10.1145/2764947.2764951. – ISBN 978-1-4503-3611-6

[van Hoorn u. a. 2012] Hoorn, André van ; Waller, Jan ; Hasselbring, Wilhelm: Kieker: A Framework for Application Performance Monitoring and Dynamic Software Analysis. In: *Proceedings of the 3rd ACM/SPEC International Conference on Performance Engineering*. New York, NY, USA : ACM, 2012 (ICPE '12), S. 247–248. – URL http://doi.acm.org/10.1145/2188286.2188326. – ISBN 978-1-4503-1202-8

[ISO 1991] ISO: Software enginnering – Product quality / International Organization for Standardization. URL https://www.iso.org/standard/16722.html, 1991 (9126:1991). – ISO

[Lilienthal 2008] Lilienthal, Carola: *Komplexität von Softwarearchitekturen, Stile und Strategien*. Von-Melle-Park 3, 20146 Hamburg, Universität Hamburg, Dissertation, 2008. – URL http://ediss.sub.uni-hamburg.de/volltexte/2008/3725

[Wettel und Lanza 2008] Wettel, Richard ; Lanza, Michele: CodeCity: 3D Visualization of Large-scale Software. In: *Companion of the 30th International Conference on Software Engineering*. New York, NY, USA : ACM, 2008 (ICSE Companion '08), S. 921–922. – URL http://doi.acm.org/10.1145/1370175.1370188. – ISBN 978-1-60558-079-1

[Yamaguchi u. a. 2014] Yamaguchi, Fabian ; Golde, Nico ; Arp, Daniel ; Rieck, Konrad: Modeling and Discovering Vulnerabilities with Code Property Graphs. In: *Proc. of IEEE Symposium on Security and Privacy (S&P)*, URL mlsec.org/joern, 2014

Hiermit versichere ich, dass ich die vorliegende Arbeit ohne fremde Hilfe selbständig verfasst und nur die angegebenen Hilfsmittel benutzt habe.

Hamburg, 09. Mai 2017 Edgar Toll