Review und Analyse von Softwarearchitekturen

Vorgehensweisen und Werkzeuge

Burkhardt Renz

Fachbereich MNI
Technische Hochschule Mittelhessen

Wintersemester 2020/21

Übersicht

- Architekturreview mit ATAM
  - Ziel der Analyse
  - Übersicht des Vorgehens
  - Erfahrungen
- Analyse von Entwürfen mit Alloy
- Architekturanalyse

Architecture Tradeoff Analysis Method

ATAM

- Methode zum Review von Architekturen
- szenariobasiert (vs. erfahrungsbasiert)
- entwickelt vom SEI an der Carnegie Mellon University

Was macht ATAM?

ATAM hilft die

- Konsequenzen der Architekturentscheidungen

in Hinblick auf

- Qualitätsmerkmale und Geschäftsziele

einzuschätzen.

ATAM-Ergebnisse

ATAM ist eine Analysemethode, die früh im Produktzyklus eingesetzt wird und folgende Punkte in einer Architektur entdecken soll

Risiken Entscheidungen, die in Zukunft Probleme bzgl. einiger Qualitätsmerkmale bereiten könnten.

Tradeoffs Entscheidungen, die ein Abwägen zwischen verschiedenen Qualitätsmerkmalen erfordern.

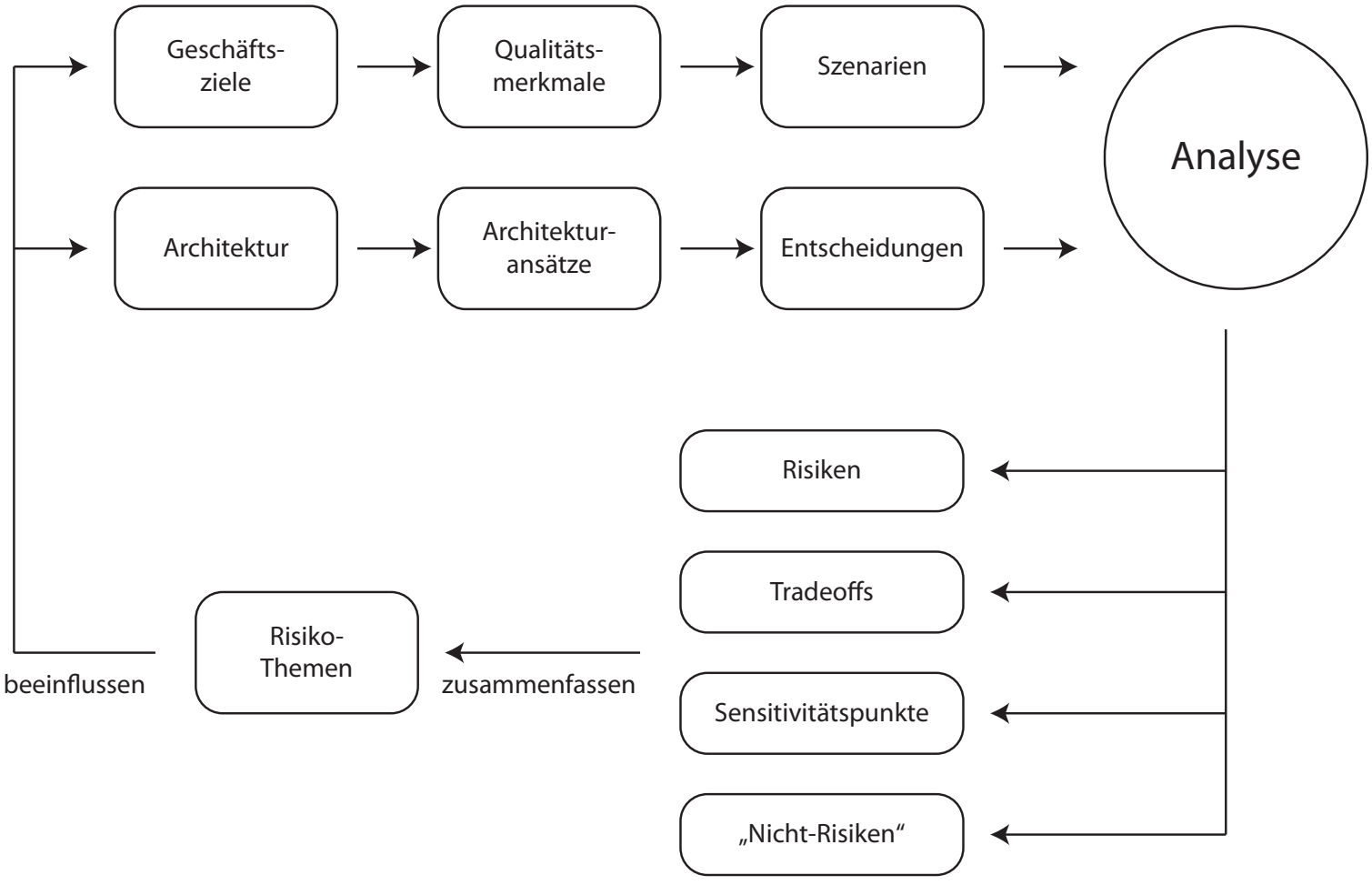
Sensitivitätspunkte Merkmale einer Architektur, die für das Erreichen bestimmer Qualitätsmerkmale entscheidend sind.

Risiko-Themen Zusammenfassung sich wiederholender Risiken, die somit einen Trend in der Architektur aufzeigen.

Nutzen

- ATAM bildet einen guten Meilenstein in der Architekturentwicklung
- Die Architekturdokumentation wird verbessert
- Risiken werden frühzeitig identifiziert
- Die geforderten Qualitätsmerkmale werden klarer
- Interessenvertreter werden über Entscheidungen, Risiken und Tradeoffs in der Architektur informiert
- Die Kommunikation zwischen den Interessenvertretern wird angestoßen oder verbessert

Übersicht des Vorgehens beim ATAM



Schritte eines ATAMs

1. ATAM vorstellen
2. Geschäftsziele präsentieren
3. Architektur präsentieren
4. Architekturansätze identifizieren
5. Qualitätsmerkmale und Szenarien erfassen (*Quality Attribute Utility Tree*)
6. Architekturansätze analysieren
7. Weitere Szenarien entwickeln und priorisieren
8. Architekturansätze analysieren (für weitere Szenarien)
9. Ergebnis präsentieren

Erfahrungen

Vorteile

- Kooperation *aller* Interessenten
- Distanz im ATAM zum Alltagsgetriebe
- Risiken erkennen $\Rightarrow$ Architektur verbessern

Nachteile

- hoher Aufwand (3 Tage Workshop – viel Vorbereitung)
- keine unmittelbaren Vorschläge als Ergebnis, nur Risikothemen

Quellen


Rick Kazman, Mark Klein, Paul Clements
ATAM: Method for Architecture Evaluation
http://www.sei.cmu.edu/library/abstracts/reports/00tr004.cfm


Rick Kazman, Mark Klein, Paul Clements
Evaluating Software Architectures: Methods and Case Studies
Boston: Addison-Wesley, 2002


Michael Stal, Stefan Tilkov, Markus Völter, Christian Weyer
SoftwareArchitekTOUR - Podcast für den professionellen Softwarearchitekten – Episode zu Architekturreviews
http://www.heise.de/developer/podcast/


Stephany Bellomo, Ian Gorton, Rick Kazman,
Toward Agile Architecture: Insights from 15 Years of ATAM Data
IEEE Software, September/October 2015

Übersicht

- Architekturreview mit ATAM
- Analyse von Entwürfen mit Alloy
  - Analysierbare Modelle
  - Beispiel
  - Analyse von Modellen
  - Einsatzmöglichkeiten
- Architekturanalyse

Modelle – und Fragen

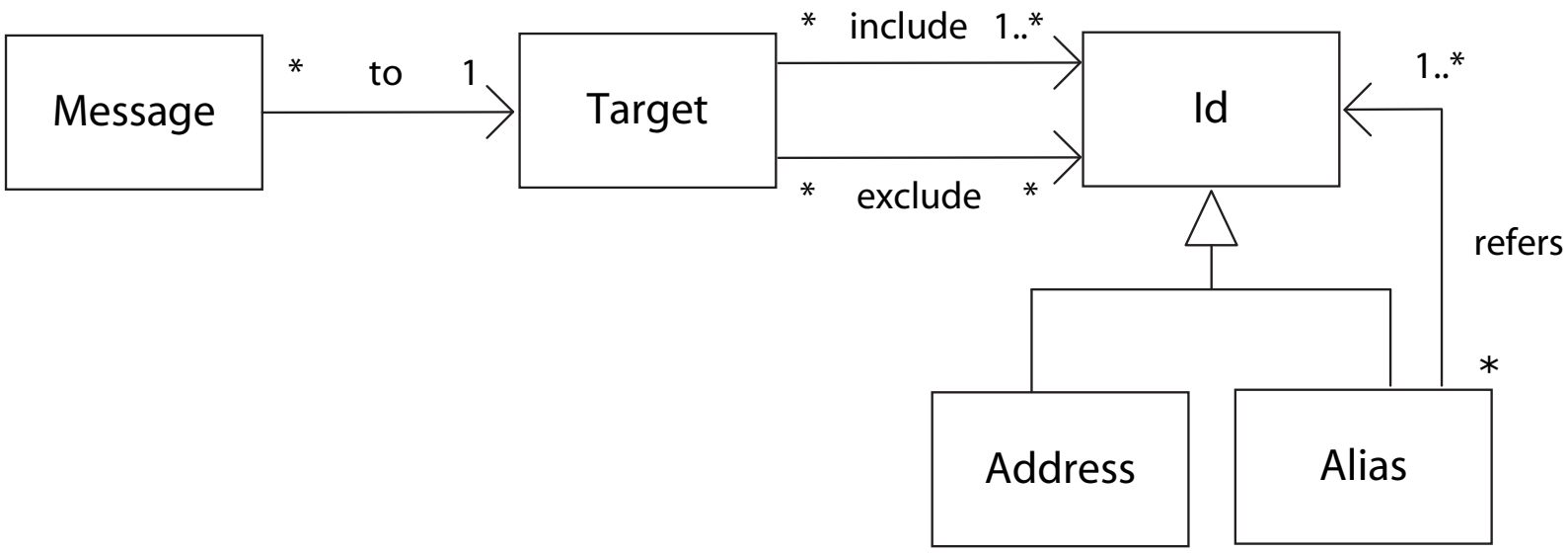
- Modelle, insbesondere UML, werden heute oft für Architektur und Design eingesetzt
- Diagramme machen Strukturen und dynamische Abläufe durchsichtiger
- Wichtige Design-Entscheidungen sind in solchen Diagrammen nicht darstellbar
- Mögliche Folge: Designfehler, die erst spät erkannt werden
- Schön wäre, man könnte Softwaredesign „ausprobieren“

Beispiel E-Mail-Programm

(nach einem Beispiel von Michael Jackson)

- Adressen *und* Aliase, die auch mehrere Adressen umfassen können
- Einer Nachricht werden nun eventuell mehrere Adressen oder Aliase als Ziel zugeordnet
- Gleichzeitig möchte man aber vielleicht bestimmte Adressen explizit als Ziel der Nachricht ausschließen – etwa private Einladung nur an Freunde, nicht an alle Kollegen
- Also machen wir ein UML-Modell:

Klassendiagramm E-Mail-Programm



. . . und nun kann man Fragen stellen

- Kann es sein, dass ein Alias sich selbst referenziert?
- Wie wird aus dieser Struktur die Menge der Adressen ermittelt, an die eine Nachricht wirklich geschickt wird?
- Zwei Strategien:
  - erst Aliase auflösen, dann ausgeschlossene wegnehmen
  - erst ausgeschlossene wegnehmen, dann Aliase auflösen
- Besteht ein Unterschied zwischen den beiden Strategien?

  Wenn ja: welche ist die gewünschte?

Spezifikation der Struktur in Alloy

```
module lfm/email

sig Message{
  to: Target
}
sig Target{
  include: set Id,
  exclude: set Id
}
sig Id{}
sig Address extends Id{}
sig Alias extends Id{
  refers: set Id
}
```

Erweiterung und Analyse der Spezifikation

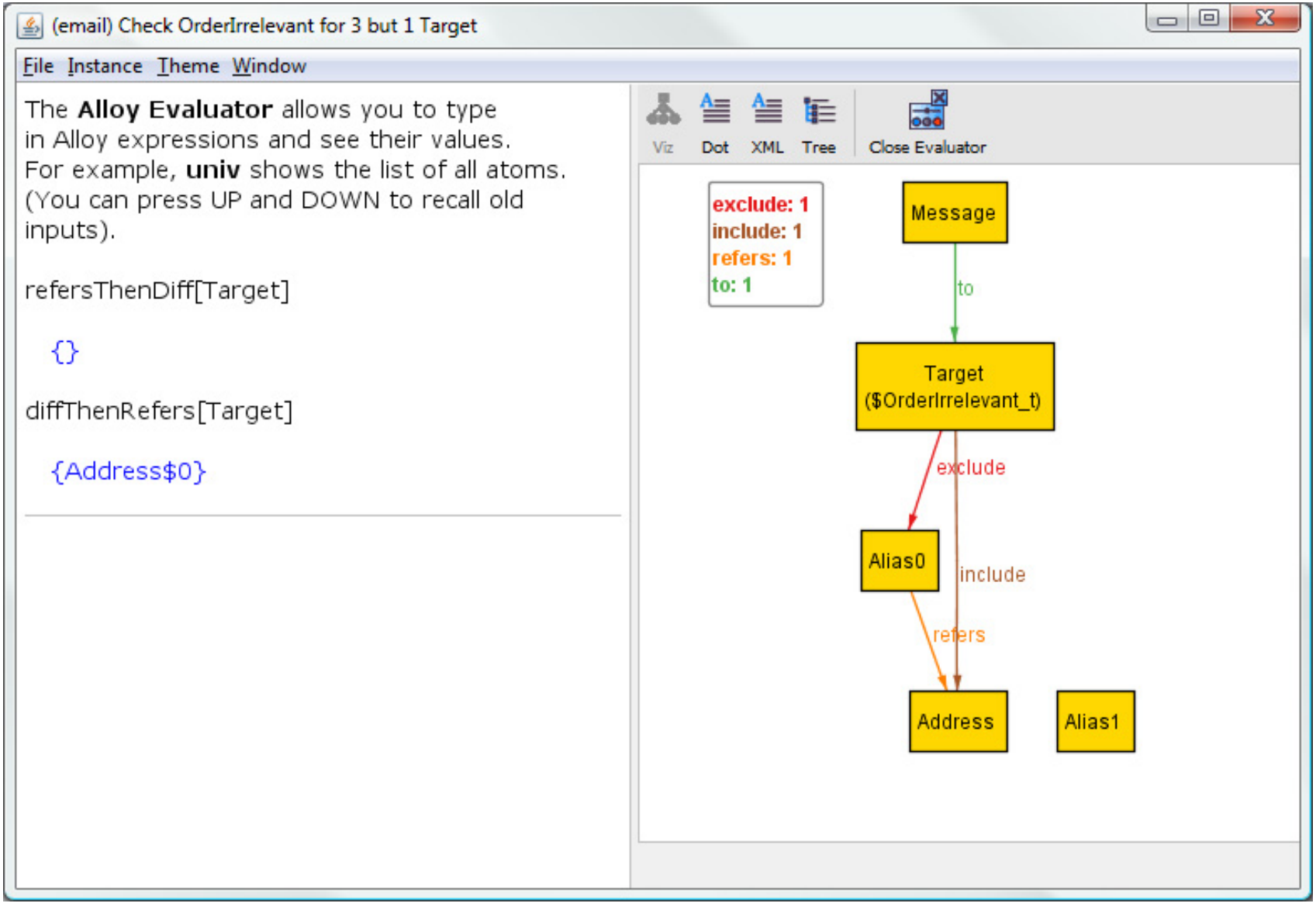
```
fact{
  no a: Alias | a in a.^refers
} // aliasing must not be cyclic

fun diffThenRefers(t: Target): set Id {
  t.(include - exclude).*refers - Alias
}
fun refersThenDiff(t: Target): set Id {
  (t.include.*refers - t.exclude.*refers) - Alias
}

assert OrderIrrelevant{
  all t: Target | diffThenRefers[t] = refersThenDiff[t]
}

check OrderIrrelevant
```

Ergebnis


(email) Check OrderIrrelevant for 3 but 1 Target
File Instance Theme Window
The Alloy Evaluator allows you to type in Alloy expressions and see their values. For example, univ shows the list of all atoms. (You can press UP and DOWN to recall old inputs).
refersThenDiff[Target]
{}
diffThenRefers[Target]
{Address$0}
Viz
Dot
XML
Tree
Close Evaluator
exclude: 1
include: 1
refers: 1
to: 1
Message
to
Target ($OrderIrrelevant_t)
exclude
include
Alias0
refers
Address
Alias1

Ergebnis

Unterschied der Strategien

- Strategie 1: (erst Dereferenzieren) Nachricht wird (in diesem Beispiel) an niemanden geschickt, weil alle eingeschlossenen Adressen auch ausgeschlossen sind
- Strategie 2: (erst Differenz bilden) Nachricht wird auch an ausgeschlossene Adressen geschickt

Ergebnis

- Die Strategien machen einen gewaltigen Unterschied
- Strategie 1 ist sicherlich die erwünschte Vorgehensweise
- Nebeneffekt: Analyse zeigt, dass eventuell gar keine Adresse übrig bleibt

Fazit

- Es gibt viele Fragestellungen, bei denen solche Techniken sinnvoll gesetzt werden können.
  - Interaktives Entwickeln von Modellen
  - Check von Spezifikationen
  - Generierung von Testfällen
  - Check von (kleinen) Klassen ggü. annotierten Schnittstellen
  - ...
- Alloy braucht eine andere Denkweise – überraschend über welch einfache Dinge man sich oft täuscht.
- Einsatz ist sehr kreativ...

Quellen

Daniel Jackson
Software Abstractions: Logic, Language, and Analysis,
MIT Press, 2011.

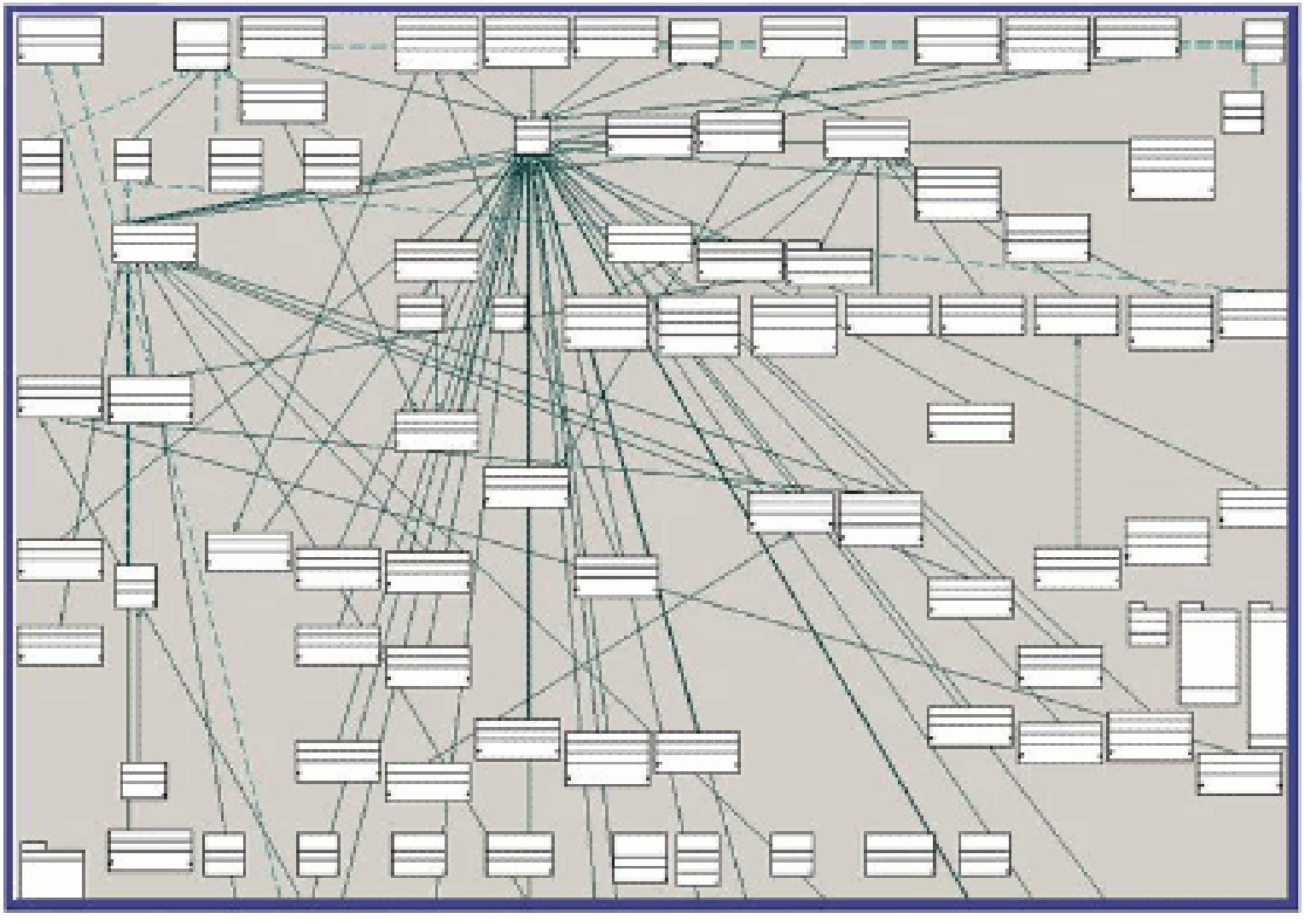
Webseite zu Alloy und dem Alloy Analyzer
`https://alloytools.org`

Michael Jackson
The Role of Structure: A Software Engineering Perspective,
in: Structure for Dependability, Springer 2006

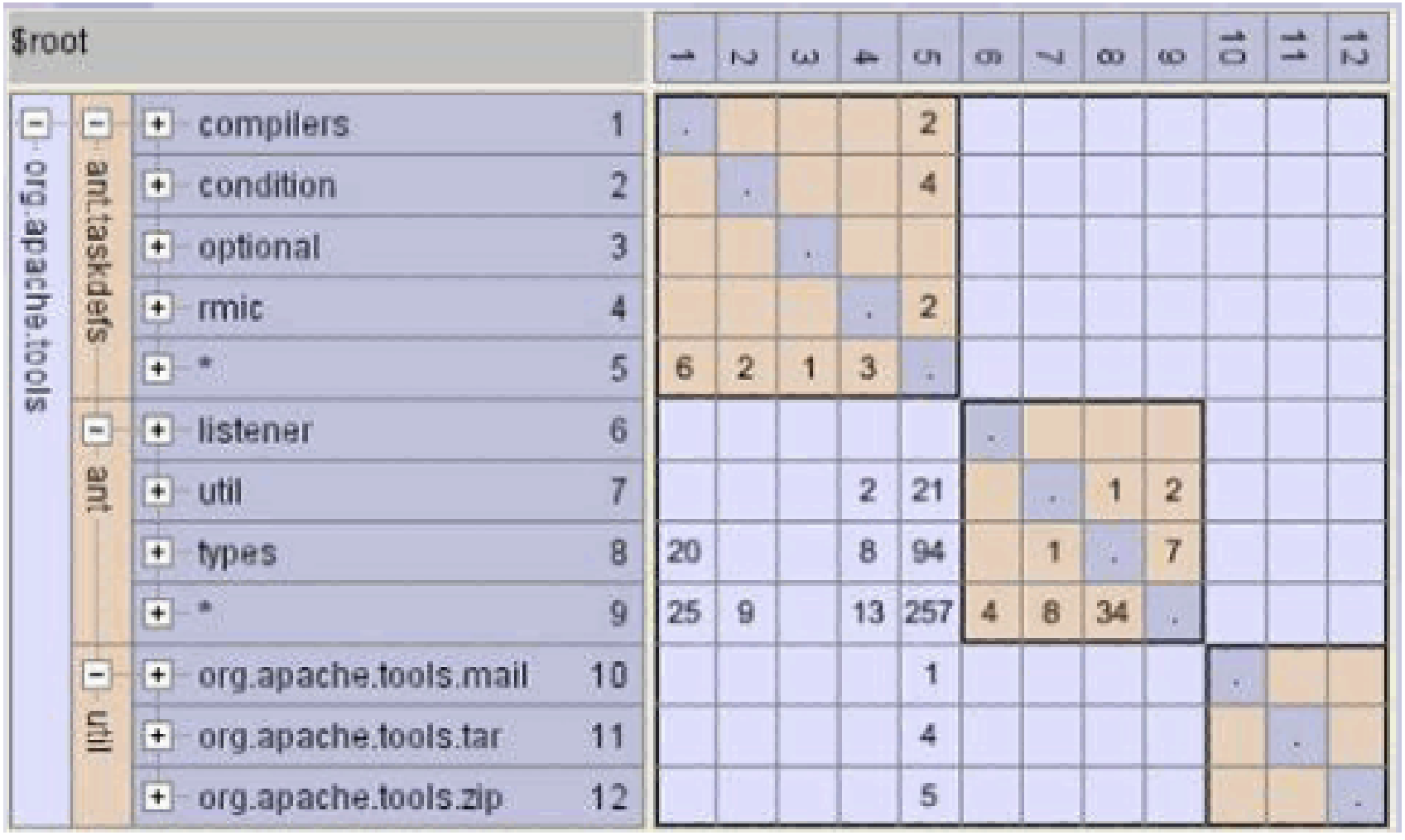
Übersicht

- Architekturreview mit ATAM
- Analyse von Entwürfen mit Alloy
- Architekturanalyse
  - Sichtbarkeit der Architektur im Code
  - Abhängigkeiten und DSM
  - Anwendung von DSM
  - Fazit

Beispiel: Das Tool ANT als UML-Klassendiagramm



Beispiel: Das Tool ANT als DSM-Diagramm



$root			1	2	3	4	5	6	7	8	9	10	11	12
org.apache.tools	ant.taskdefs	compilers	1	.			2							
		condition	2		.		4							
		optional	3			.								
		rmic	4				.	2						
		*	5	6	2	1	3	.						
	ant	listener	6						.					
		util	7				2	21		.	1	2		
		types	8	20			8	94		1	.	7		
		*	9	25	9		13	257	4	8	34	.		
	util	org.apache.tools.mail	10					1					.	
		org.apache.tools.tar	11					4						.
		org.apache.tools.zip	12					5						.

Geeignete Darstellung der Code-Struktur?

UML-Strukturdiagramme:

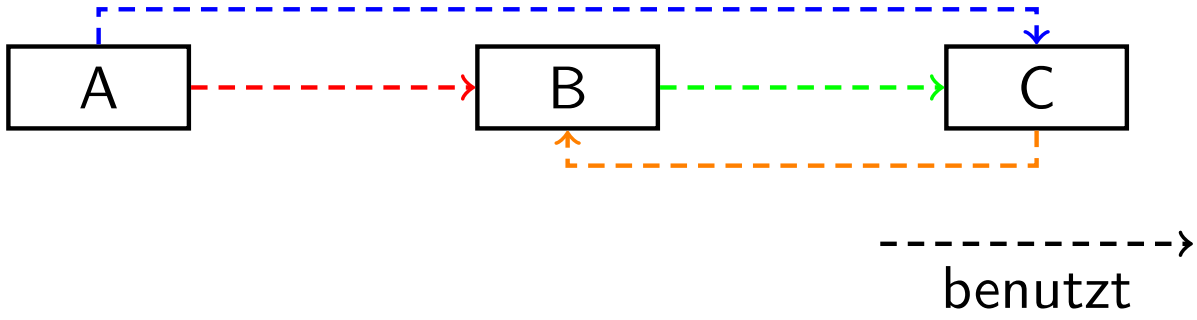
- \+ viel Detailinformation
- \+ unterscheidet Arten der Abhängigkeit
- – viele Komponenten (Klassen, Pakete, . . .) $\Rightarrow$ unübersichtlich

Dependency Structure Matrix (DSM, $\approx$ 1970):

- – zunächst ungewohnt
- \+ Detailinformation versteckt
- \+ viele Komponenten (Klassen, Pakete, . . .) $\Rightarrow$ übersichtlich

Erzeugen einer DSM

- Abhängigkeit zwischen Komponenten

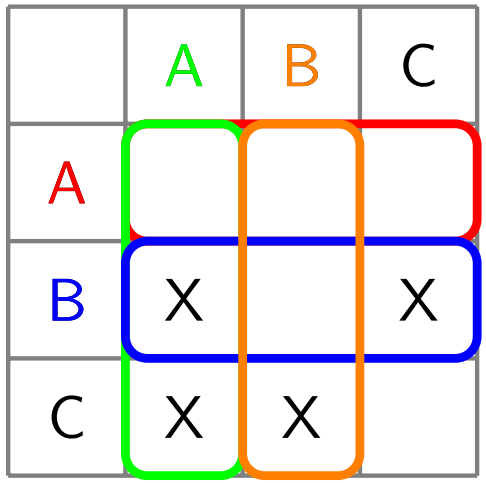


- Resultierende DSM

	A	B	C
A			
B	X		X
C	X	X	

Lesen einer DSM

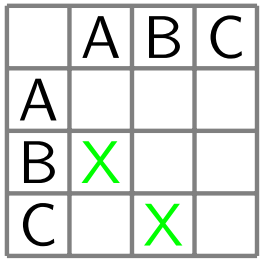
- DSM



- Interpretation
  - Zeile A: A wird von keiner Komponente benutzt
  - Spalte A: A benutzt B und C
  - Zeile B: B wird von A und C benutzt
  - Spalte B: B benutzt C

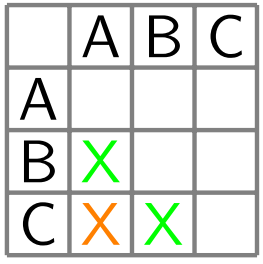
DSM-Archetypen

- Schichtenarchitektur



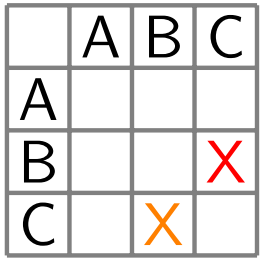
	A	B	C
A			
B	X		
C		X	

- keine zirkuläre Abhängigkeit



	A	B	C
A			
B	X		
C	X	X	

- zirkuläre Abhängigkeit



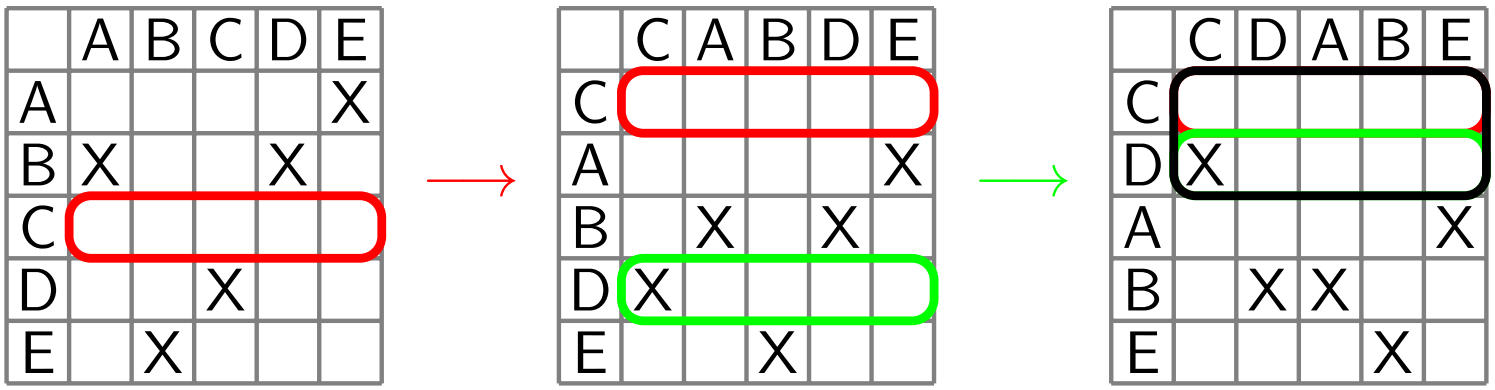
	A	B	C
A			
B			X
C		X	

Anwendungsfall „Tatsächliche Architektur ermitteln“

- Input: DSM
- DSM umsortieren
- Komponenten zusammenfassen
- Output: DSM

DSM umsortieren

Input:



Ziel: keine Einträge oberhalb der Diagonale

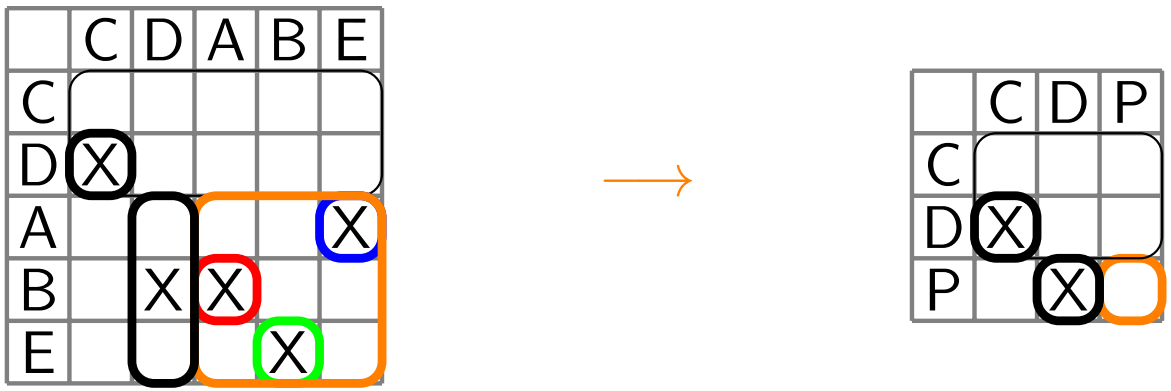
1. Schritt: C wird von keiner Komponente benutzt
   $\Rightarrow$ C ist 1. Schicht

2. Schritt: D wird nur von C benutzt
   $\Rightarrow$ D ist 2. Schicht

Resultat: Subdiagonalform für C und D

Komponenten zusammenfassen

Input:



Ziel: Zyklus eliminieren

1. Schritt: Zyklus erkennen: A→B→E→A
   ⇒ A, B, E bilden einen Zyklus
2. Schritt: neue Komponente: P = {A, B, E}
   ⇒ neue DSM

Resultat: Subdiagonalform = Schichtenarchitektur

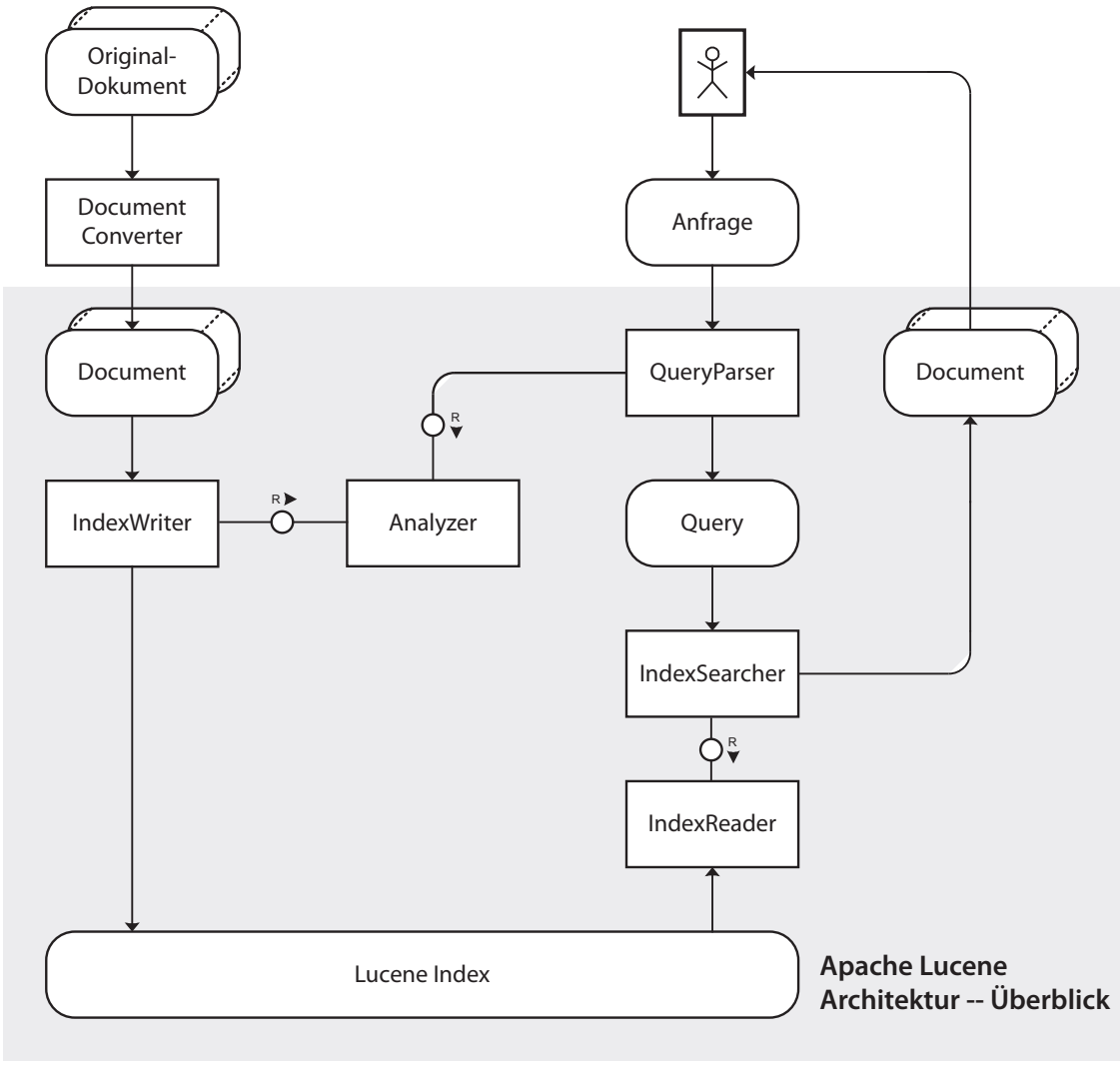
Anwendungsfälle

- tatsächliche Architektur ermitteln
- Paket-Struktur an tatsächliche Architektur anpassen
  (Output: notwendige Code-Änderungen)
- Architektur festlegen und regelmäßig überprüfen
- externe Code-Abhängigkeiten verwalten
- Redundanzen eliminieren
- ...

MNI Macro Processor MMP

Demo – siehe Extra-Video saa-rev-mmp.mp4

Beispiel Apache Lucene



Apache Lucene

Demo – siehe Extra-Video saa-rev-luc.mp4

Fazit

- UML als Mittel der Darstellung und Analyse der Architektursicht des Codes nicht geeignet
- DSM-basierte Werkzeuge gestatten die Abhängigkeitsanalyse
- Sie ermöglichen es auch, die Entwicklung der Architektur zu überwachen
- Heute ohne großen Aufwand einsetzbar

Quellen


The Design Structure Matrix (DSM) Homepage
`http://www.dsmweb.org`


Webseite von Lattix, Inc.
`http://www.lattix.com`


WebSeite zu Sonargraph
`http://www.hello2morrow.com`


Petra Becker-Pechau, Bettina Karstens, Carola Lilienthal
Automatisierte Softwareüberprüfung auf der Basis von Architektur Regeln
in: Software Engineering 2006, Springer LNI P-79, 2006


Carola Lilienthal
Langlebige Softwarearchitekturen: Technische Schulden analysieren, begrenzen und abbauen
dpunkt.verlag, 2019