Softwareanalyse

Einsatzgebiete, Werkzeuge und Beispiele

Bodo A. Igler und Burkhardt Renz

Fachhochschule Gießen-Friedberg

Workshop VfSt 17. August 2006

- Analysierbare Modelle
  - Modelle – und Fragen
  - Analyse von Modellen
  - Konsequenzen
- Architekturanalyse
  - Sichtbarkeit der Architektur im Code
  - Architekturanalyse mit DSM
  - Konsequenzen
- Codeanalyse
  - Test – Reviews – Codeanalyse
  - Werkzeuge zur Codeanalyse
  - Konsequenzen

Modelle – und Fragen

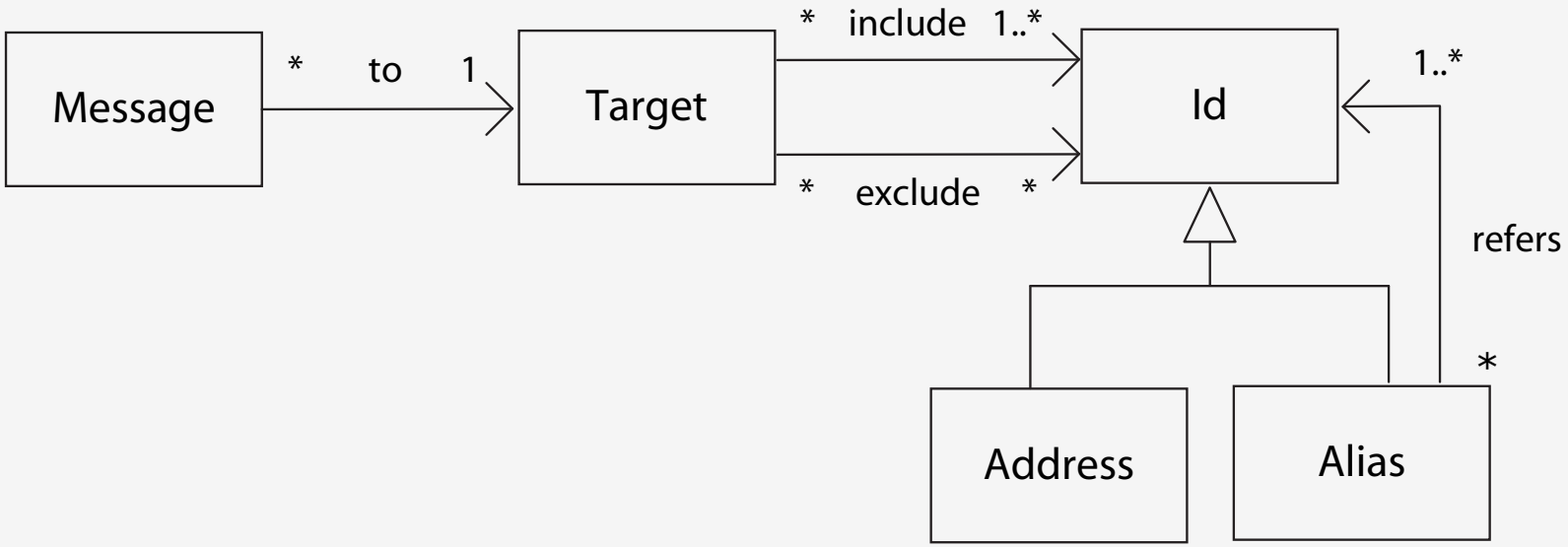
- Modelle, insbesondere UML, werden heute oft für Architektur und Design eingesetzt
- Diagramme machen Strukturen und dynamische Abläufe durchsichtiger
- Aber: sie suggerieren manchmal Sachverhalte, die sie gar nicht enthalten
- Wichtige Design-Entscheidungen sind in solchen Diagrammen nicht darstellbar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

Beispiel E-Mail-Programm

(nach einem Beispiel von Michael Jackson)

- Design eines E-Mail-Programms
- 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

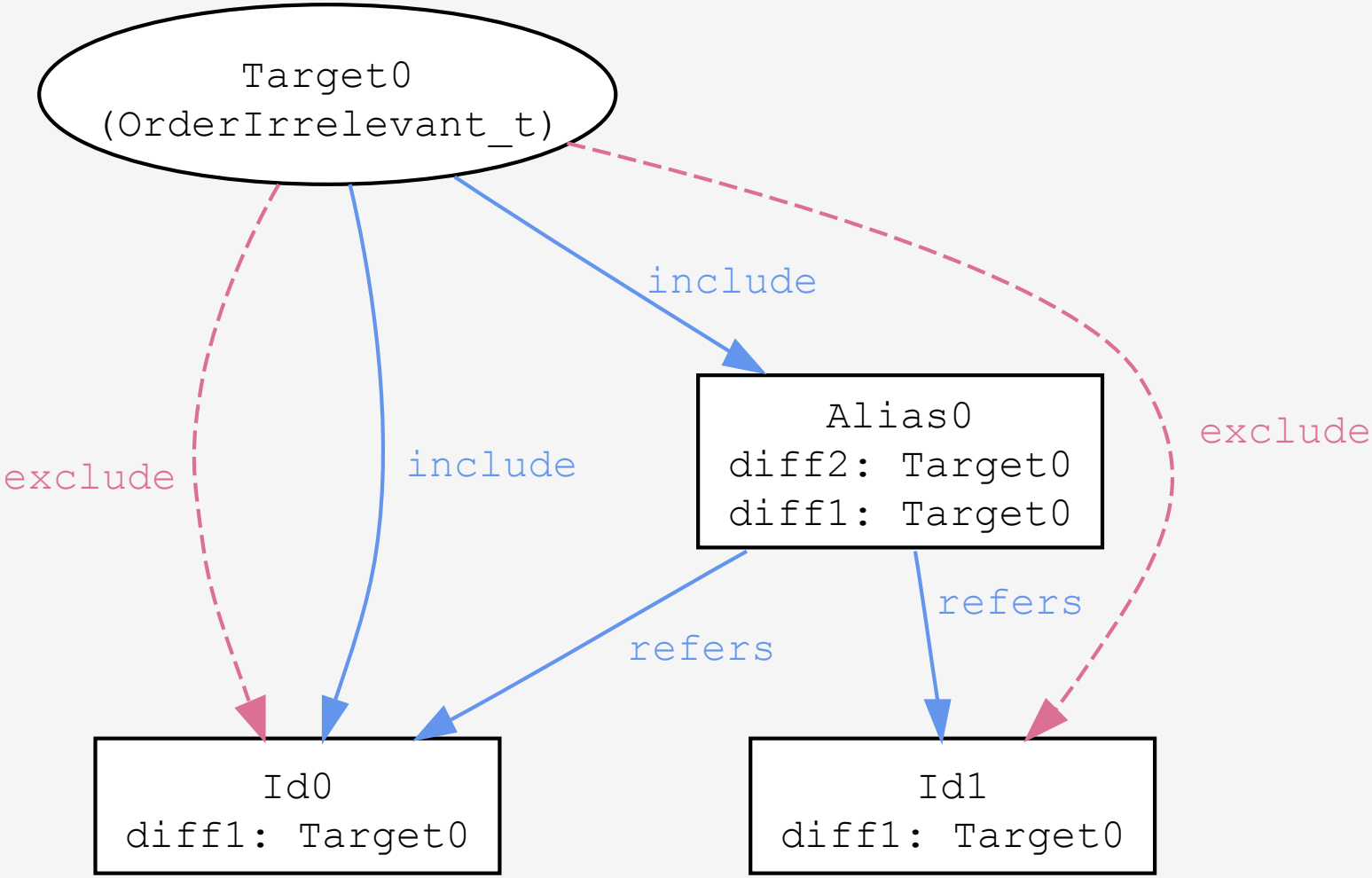
```
/* signatures */
fact{
  no a: Alias | a in a.^refers
} // aliasing must not be cyclic

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
```

Demo

Ergebnis


Target0
(OrderIrrelevant_t)
include
exclude
include
exclude
Alias0
diff2: Target0
diff1: Target0
refers
refers
Id0
diff1: Target0
Id1
diff1: Target0

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

Fazit

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

Konsequenzen

- Gibt es Fragestellungen, bei denen solche Techniken sinnvoll einsetzbar sind?
- Gibt es Probleme in der aktuellen Software, die man mit Softwareanalyse angehen sollte?
- Gibt es Modelle komplexer Strukturen, die man einfach probeweise mit Alloy analysieren könnte?
- ...

Quellen

Daniel Jackson
Software Abstractions: Logic, Language, and Analysis,
MIT Press, 2006.

Webseite zu Alloy und dem Alloy Analyzer
`http://alloy.mit.edu`

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

Geeignete Darstellung der Code-Struktur?

UML-Strukturdiagramme:

- \+ viel Detailinformation
- \+ Abhängigkeitsgraph = intuitive Form der Darstellung
- – viele Komponenten (Klassen, Pakete, . . .) ⇒ unübersichtlich

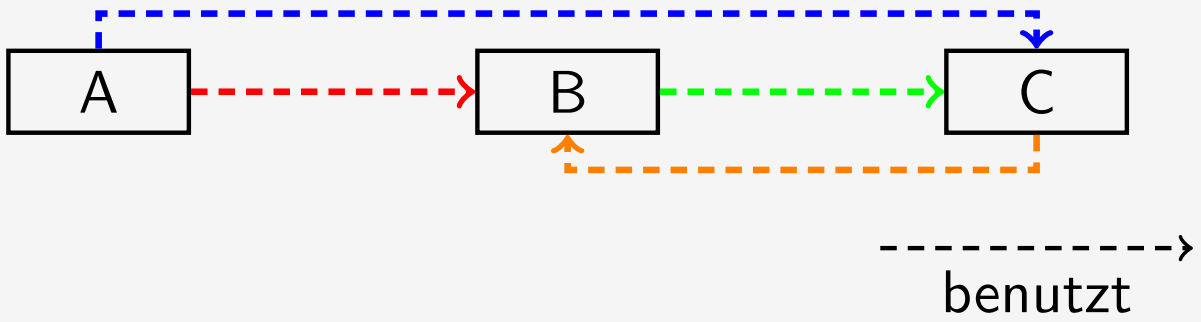
Design Structure Matrix (DSM, ≈ 1970):

- – keine Detailinformation
- – zunächst ungewohnt
- \+ viele Komponenten (Klassen, Pakete, . . .) ⇒ übersichtlich

Frage: Was ist eine DSM?

Erzeugen einer DSM

- Abhängigkeit zwischen Komponenten

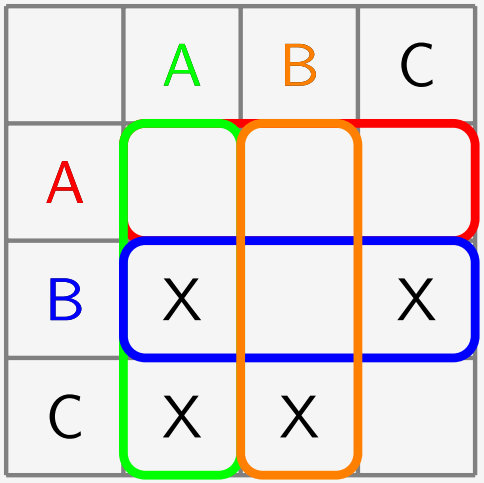


- Resultierende DSM

	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

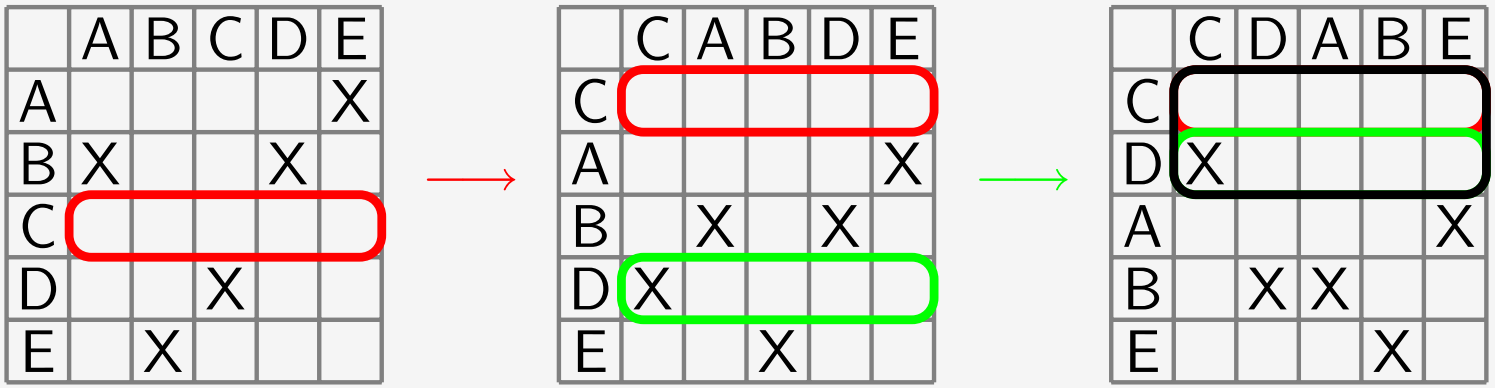
	A	B	C
A			
B			X
C		X	

Anwendungsfall „Tatsächliche Architektur ermitteln“

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

DSM Umsortieren

Input:



Ziel: keine Einträge oberhalb der Diagonale

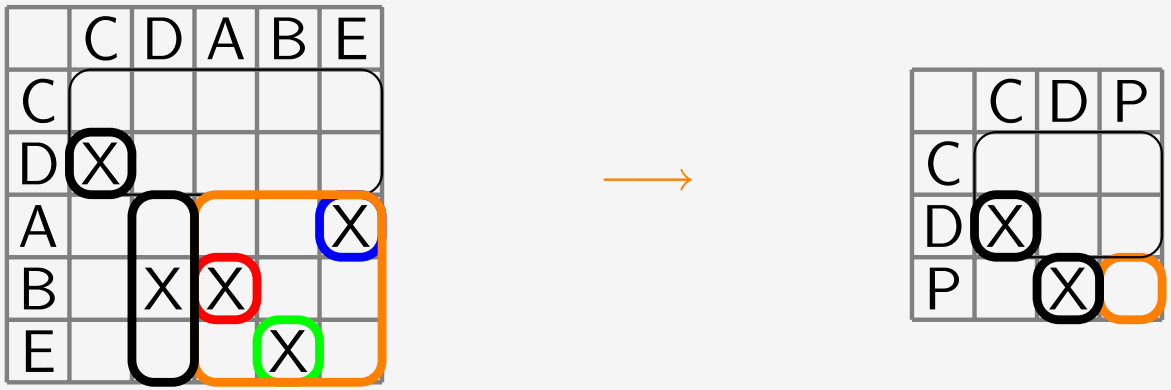
1. Schritt: C wird von keiner Komponente benutzt
   $\Rightarrow$ C ist 1. Schicht

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
- ...

Beispiel: *LATTIX LDM*

- Input = JAVA, C++, ...
- Darstellung:
  - DSM
  - Block-Struktur
  - Auf-/Zuklappen verschachtelter Komponenten
  - Detailinformation in separatem Fenster
- Benutzerabhängige Definition:
  Abhängigkeit := Methodenaufruf, Vererbung, ...
- Metriken

Konsequenzen

- Sollte der Einsatz solcher Werkzeuge verstärkt werden?
- Lohnt es sich Werkzeuge für die Architekturanalyse zu evaluieren?
  - JDepend
  - LATTIX LDM
  - Sotograph
  - ...
- Kann man sich von einer Analyse neue Erkenntnisse über die Architektur des Systems erwarten?
- Gibt es Subsysteme, bei denen sich genauere Analyse lohnt?
- ...

Quellen

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

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

WebSeite zu Sotograph
`http://www.sotograph.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

Tests – Reviews – Codeanalyse

Tests

- (Automatisierte) Unittest erhöhen Softwarequalität erheblich
- Integrationstests benötigen starke Pfadüberdeckung

Reviews

- Reviews entdecken Fehler, die man durch Tests nicht findet
- Reviews schaffen eine Qualitätskultur

Codeanalyse

- Und der schwierige Rest?
- insbesondere z.B. Fragen der Nebenläufigkeit

Demo

Beispiel Java PathFinder

- Java PathFinder ist ein Modelchecker für Java Bytecode
- Konzept: Ein Programm wird gesehen als eine Folge von Zuständen, die durch Anweisungen verändert werden
- Technik: JPF ist eine eigene virtuelle Maschine, die Modelchecking beherrscht.
- Konsequenz: JPF prüft *alle* möglichen Pfade durch ein Java-Programm

Einsatzmöglichkeiten und -grenzen

Einsatzmöglichkeiten

- Deadlocks, unerwartete Exceptions
- erweiterbar durch eigene Listener

Grenzen

- nur reiner Java Bytecode
- Code $<$ 10 kLoC, d.h. eigene Testrahmen erforderlich

Konsequenzen

- Treten Probleme mit Nebenläufigkeit auf?
- Gibt es im Test unentdeckte Low-Level-Fehler, die zur Laufzeit auftreten?
- Gibt es wiederkehrende Probleme, die einem bestimmten Muster folgen (wie früher „der wandernde Fehler“)?
- Sollte man Werkzeuge wie etwa JPF, JLint,... ausprobieren?
- Sind Tools für die Testfall-Generierung sinnvoll?
- Ist eine Analyse sinnvoll oder nötig, worin potentielle Schwachstellen liegen? Oder eh' alles im Griff?
- ...

Quellen

Webseite zu Java PathFinder
`http://javapathfinder.sourceforge.net`

W. Visser, K. Havelund, G. Brat, S. Park und F. Lerda
Model Checking Programs
in: Automated Software Engineering Journal 10(2), 2003

WebSeite zu JLint
`http://jlint.sourceforge.net`

Cyrille Artho
Finding faults in multi-threaded programs
Thesis ETH Zürich 2001