

Impact Analysis für AOP

Maximilian Störzer
Lehrstuhl Softwaresysteme
Universität Passau



Inhalt

Inhalt - kurze Übersicht

- AOP und AspectJ, ein Überblick
 - Was verbirgt sich hinter AOP?
 - Ein Prototyp: AspectJ
 - Probleme bei aktuellen Sprachen
- aktuelle Forschungsprojekte
 - Berechnung von Pointcut-Deltas
 - Deklarative Pointcuts
 - **Statische und dynamische Impact Analysis**

Guter Entwurf, Grundsätze des SE

Kurzer Überblick: Grundsätze beim Softwareentwurf.

Module sollen

- intern eine *hohe Kohäsion* aufweisen
- untereinander *schwach gekoppelt* sein.

Ziel: Separation of Concerns – gute Modularität, erlaubt

- *Lokalisierung von Änderungen* → Wartbarkeit
- separate *Wiederverwendung* der Module.

Was verbirgt sich hinter AOP_____

Entwicklung des Software Engineering

Separation of Concerns ist ein Leitfaden
bei der Entwicklung des Software Engineering:

1. **Geheimnisprinzip** (Modul, ADO)
2. **Instantiierung** (ADT)
3. **Varianten und Polymorphismus** (OO, Templates)

OO ist derzeit *state-of-the-art*, aber ...

Optimaler Entwurf oft nicht erreichbar

Häufig wird von Software-Systemen die Erfüllung sogenannter *“non-functional requirements”* gefordert.

Beispiele: Logging/Tracing, Persistenz, Authorisierung, Verteilung, ...

Derartige Anforderungen sind im OO-Paradigma häufig *kaum modularisierbar*.

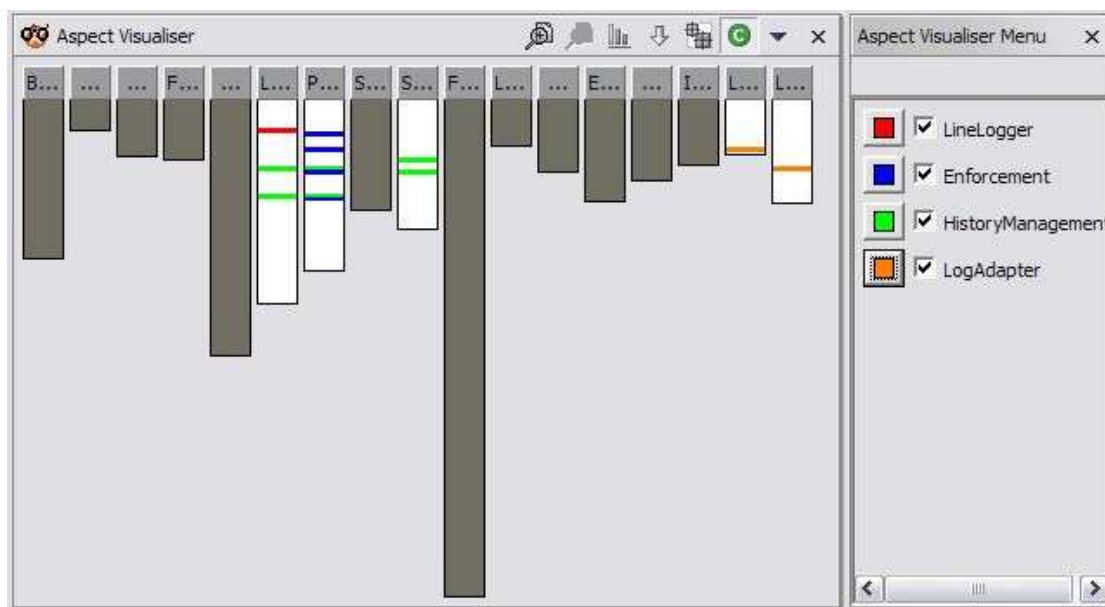
Weiterer Gesichtspunkt: Systemevolution.

Falls neue Anforderungen nicht zum Entwurf passen, resultieren Änderungen *im ganzen System (invasive changes)*.

Nicht-modularisierbare Anforderungen werden als *Crosscutting Concerns* bezeichnet.

Was verbirgt sich hinter AOP

Beispiel für Crosscutting Concerns



Ein Beispiel für Crosscutting Concern (aus AJDT Manual).

AOP als mögliche Lösung

“The next big thing after OO!”

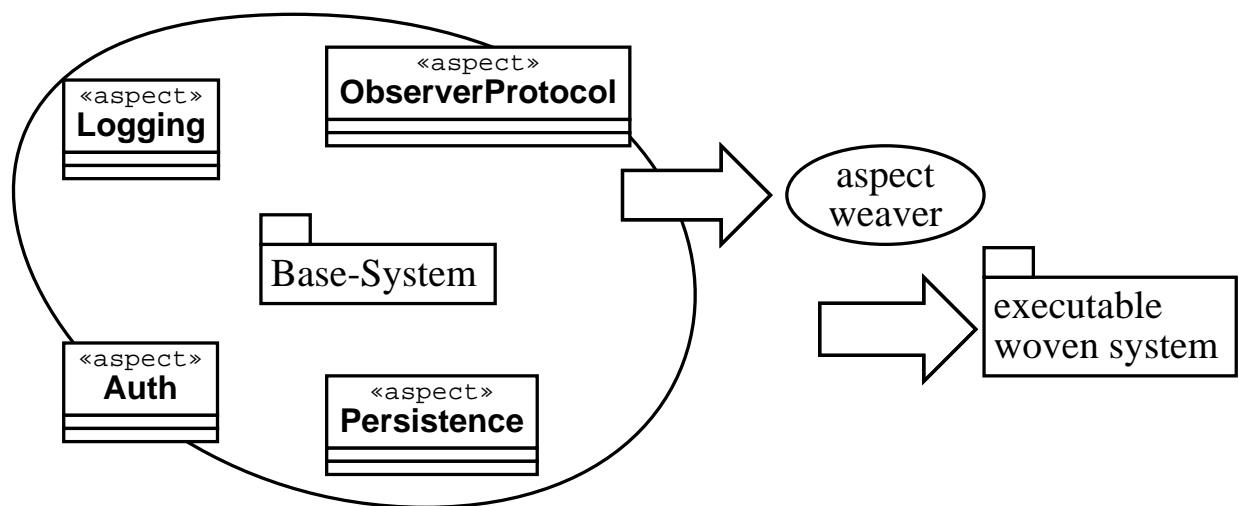
AOP erlaubt *Modularisierung von Crosscutting Concerns*:

- **Idee:** *Aspekt* als neue Art von Modul kapselt Crosscutting Concerns
- Aspekt definiert **wo was** passiert:
→ *pointcuts* und *advice*.
- Compiler oder *Aspect-Weaver* kombiniert Funktionalität mit traditionellen Basis-System.

Derzeit oft als Erweiterung für OO-Sprachen (AspectJ, AspectC++, AspectS, ...), *nicht* auf OO-Paradigma beschränkt.

Was verbirgt sich hinter AOP _____

Weaving – Schematische Darstellung



Ein Prototyp: AspectJ

AOP wurde u.a. durch die Sprache AspectJ als Erweiterung von Java implementiert.

Neue Sprachkonstrukte:

inter type declarations (ITDs): Open Classes für Java, erlaubt Erweiterung einer Klasse um neue Members (Felder und Methoden).

pointcuts: Legt fest, *wo* ein Aspekt Funktionalität des Basissystems modifiziert.

advice: Legt ähnlich einer Methode fest, *wie* die Funktionalität des Basissystems modifiziert wird.

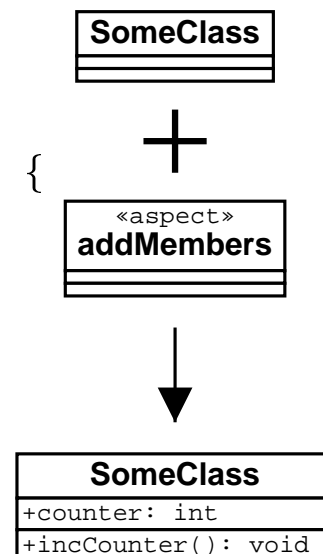
Ein Prototyp: AspectJ

Hinzufügen neuer Elemente zu Klassen: ITDs

```
aspect AddMembers {  
    int SomeClass.counter;  
    void SomeClass.incCounter() {  
        counter++;  
    }  
    ...  
}
```

Achtung: Sichtbarkeit ist *auf den Aspekt* bezogen!

Neu eingefügte Members können dann vom Aspect-Code aus benutzt werden.



Ein erster Aspekt: Tracing

```
aspect TraceFoo {
    pointcut callFoo(): call(public foo(int))
        && !within(TraceFoo);

    before(): callFoo() {
        String sig = thisJoinpoint.getSignature();
        System.out.println(sig);
    }
}
```

Parameterwerte sind z.B. via Reflection zugänglich.

Ein Prototyp: AspectJ

Fazit: AspectJ

AspectJ ist eine interessante Erweiterung von Java

Development Aspects: Naheliegender Einsatz als
Entwicklungstool – Debugging durch Tracing in 10 Zeilen!

Productive Aspects: Aspekte sind gut geeignet für
“non-//funktional requirements”: Authorisierung, Persistenz,
Verteilung, Logging, ...

Im Gegensatz zu den meisten anderen AO-Sprachen stehen
mit den *AspectJ Development Tools* für AspectJ gute
Entwicklungswerkzeuge (Eclipse Integration) zur Verfügung.

Probleme bei aktuellen Sprachen

AOP hat noch einige relevante Probleme:

- SE Sicht: Aufhebung Geheimnisprinzip!
- Hohe Kopplung zwischen Aspect und Basis-System, “fragile” Pointcuts
- **Semantik: Auswirkungen von Aspekten oft unklar**

Aspekte als mitgelieferte Source-Patches?

Probleme bei aktuellen Sprachen

Kopplung Aspect - Basis-System

Pointcuts beziehen sich *explizit* auf z.B. Methoden des Basis-Systems:

```
pointcut traceMySetters():  
    ( call(* setFoo(..)) ||  
      call(* setBar(..)) ||  
      ... )  
    && within(myProject);
```

Dadurch wird die *Wiederverwendung* des Aspektes mit anderem System *erschwert*.

Ver(schlimm)besserung: Wildcards

Anstatt jeden Anwendungsstelle explizit aufzuführen, werden *Wildcards* verwendet:

```
pointcut traceMySetters():  
    call(* set*(..)) && within(myProject);
```

Problem: Vertrauen auf *naming conventions*, deren Einhaltung nicht garantiert ist.

Besonders an Pointcutsprachen wird derzeit gefeilt:

strukturbasierte pointcuts: Identifizierung über Typen

deklarative pointcuts: Identifizierung über semantische Eigenschaften

Probleme bei aktuellen Sprachen

Fragile Pointcuts

Wildcard- und Name-based Pointcuts sind problematisch:

→ Semantik durch *triviale non-lokale Änderungen* massiv beeinflussbar!

Source-Modifikationen wie

- Umbenennung/Hinzufügen/Löschen von (private!) Methoden/Feldern/Klassen
- kleine Änderungen an Pointcuts

ändern die Ergebnisse von Pointcuts und damit die Ausführung von Advice, d.h. die Semantik des Systems!

Ermittlung der Auswirkungen von Programmänderungen auf Pointcuts

Unterschied zu normalen Umbenennungen: Der Compiler antwortet mit einer Fehlermeldung, falls nun ein Name nicht gebunden ist, der Weaver nicht!

Idee: Analyse der Änderungen bei Pointcuts zwischen verschiedenen Programmversionen, Anzeige im Editor.

→ Der Programmierer wird *vom System* auf die geänderte Semantik der Aspekte hingewiesen, auch wenn kein Aspekt bearbeitet wird!

*Derzeit laufendes Programmierpraktikum.
Ziel ist ein Eclipse-Plugin, das ajdt erweitert.*

Declarative Pointcuts am Beispiel des Observer Patterns

Declarative Pointcuts am Beispiel des Observer Patterns

Observer-Pattern ist ein wichtiges Patterns, um z.B. eine GUI unabhängig vom zugrundeliegenden Modell zu machen.

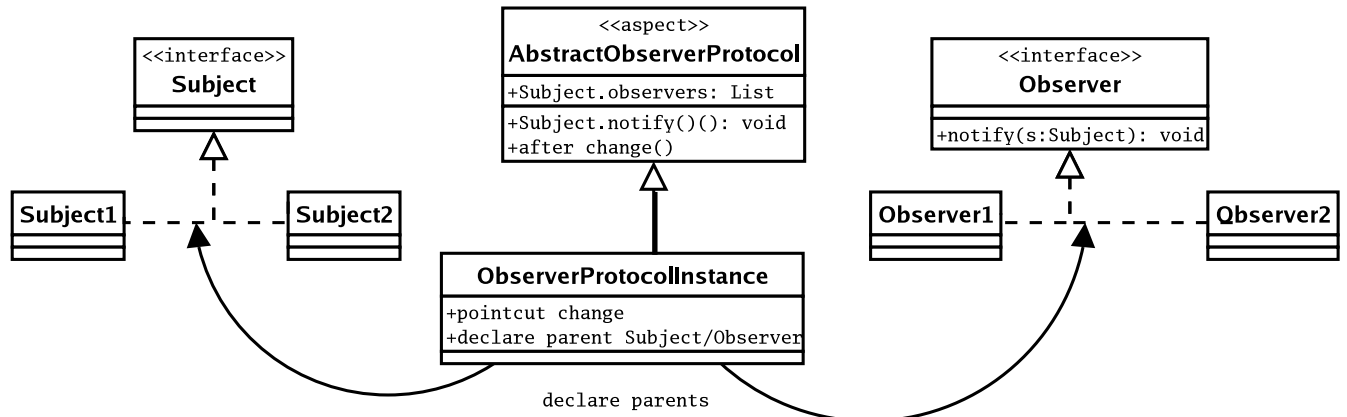
Dennoch: Das Modell muss für die Observierung vorbereitet werden (`notify()`-calls, Verwaltung der Observer)
→ keine Aufgabe des Subjektes

Idee: Subjekte wissen gar nicht mehr, dass sie observiert werden. Kapselung des Subjekt-Observer-Protokolls im Aspekt.

Voraussetzung: Observer kann Subjekte verarbeiten, Verarbeitung wird durch den Aspekt angestossen.

UML für ein AOP-Observer-Pattern

Neues Konzept hier: Vererbung von Aspekten. Erlaubt Redefinition von *abstrakten Pointcuts*.



Declarative Pointcuts am Beispiel des Observer Patterns

Der abstrakte Aspekt AbstractObserverProtocol

```
abstract aspect AbstractObserverProtocol {  
    // Verwaltung der Observer  
    public List Subject.observers;  
    public void Subject.notify() ...  
  
    // Reaktion auf Modelländerungen  
    abstract pointcut change();  
    after(Subject s): change() && this(s) {  
        s.notify();  
    }  
}
```

Konkrete Instanz: `ObserverProtocolInstance`:

```
abstract aspect ObserverProtocolInstance
  extends AbstractObserverProtocol {

  pointcut change():
    set(Subject1.x) || set(Subject2.y);

  declare parents Subject1, Subject2
    implements Subject;
  declare parents Observer1, Observer2
    implements Observer;
}
```

Declarative Pointcuts am Beispiel des Observer Patterns

Wertung dieses Modells ...

Vorteile:

- Beliebige Subjekte können observiert werden, deren Code wird nicht mehr durch `notify()`-calls bzw. die Observer-Verwaltung "verschmutzt".
- Generisches Protokoll *vollständig* im Aspekt gekapselt.
- Instanziierung durch geeigneten abgeleiteten Aspekt macht *Aspekt wiederverwendbar*.

Nachteile:

- Der abgeleitete Aspekt ist immer noch stark an das Basissystem gekoppelt.
- `notify()`-calls ggf. "unschärfer".
- Der Pointcut "change" ist fragile.

Idee der deklarativen Pointcuts

Spezifiziere Pointcuts nicht über lexikalische oder strukturelle Constraints, sondern über *semantische Eigenschaften*.

Beispiel (Keynote, Kizcales AOSD03):

```
// berechne alle relevanten Felder
pointcut* accessedByNotify():
    pcfFlow(execution(void Obersver+.notify(..))
        && get(* Subject+.*));
// update(), falls sich eines davon ändert
after(): set(<accessedByNotify(>) {
    notify(); // call für relevante Subjekte
}
```

Declarative Pointcuts am Beispiel des Observer Patterns

Idee der deklarativen Pointcuts (2)

Das ist generell für das Observer-Pattern und auch in anderen Kontexten anwendbar.

Hintergrund:

- Man stellt *semantisch höherwertige Pointcuts* zur Verfügung.
- Zur Compilezeit werden diese ausgerechnet.

Vorteil: Beim Observer-Pattern könnte man den change-Pointcut *in den abstrakten Aspekt verschieben* (!).
Instanziierung bedeutet dann nur noch, Subjekte und Observer anzugeben.

Derzeit laufende Diplomarbeit: Erstellung einer Analyse zur Berechnung von pcfFlow(..) als AspectJ Spracherweiterung.

Forschungs-Kerngebiet: Impact Analysis

Setting:

- Basissystem B
- Menge von Aspekten $\mathcal{A} = \{A_1, \dots, A_k\}$
- Resultierendes System: $S = B \oplus \mathcal{A}$

Kernfragen:

*Was stellen diese Aspekte mit meinem System an?
Wie testet man derartige Systeme?*

AOP Grundeigenschaft: *Obliviousness*

Eine Kern-Eigenschaft von AOP: *Obliviousness*, d.h. Base-Code ist sich *nicht* der Modifikation durch Aspekte bewusst.

Vorteil: Base muss nicht auf Modifikationen vorbereitet werden: Kernvoraussetzung, um invasive Changes zu vermeiden.

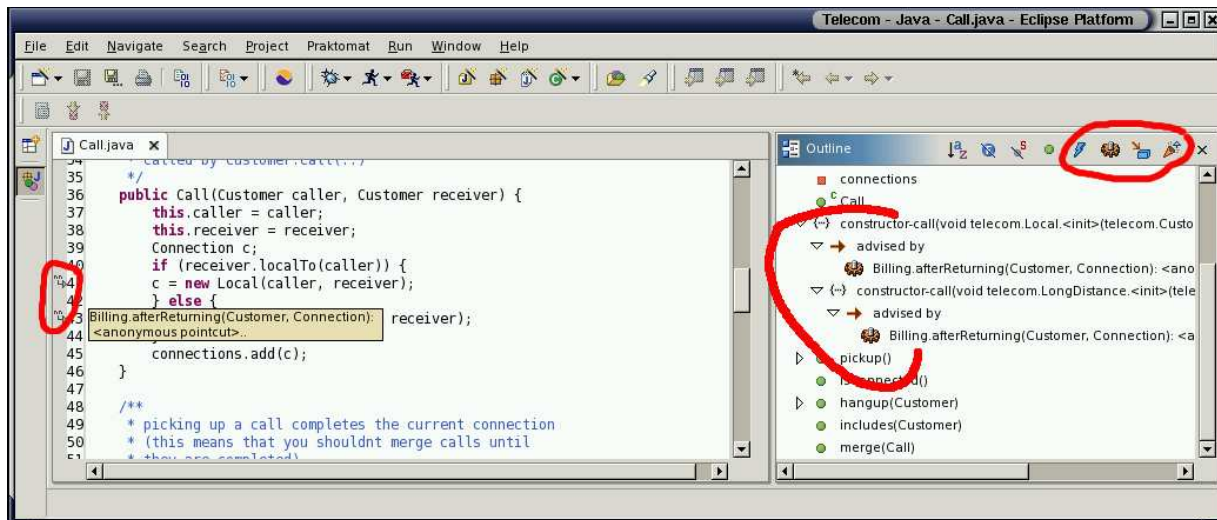
Nachteil: Die Semantik eines Moduls ist nicht mehr nur dessen Quellcode zu entnehmen, *alle möglicherweise relevanten Aspekte* müssen berücksichtigt werden.

Verlust des “modular reasoning”!!

Diese Problem ist bekannt, Antwort: *Tool Support*

Beispiel für Tool Support: *ajdt*

Die *AspectJ development Tools (ajdt)* berechnen, wo advice angewendet wird.



Nur syntaktischer, kein semantischer Support!

Impact Analysis für AOP

Offene Fragen

Ajdt als Antwort auf die Obliviousness-Problematik ist wichtig, aber

- gibt (derzeit) *keine Unterstützung*, ob sich Pointcut-Mengen durch Systemmodifikationen *geändert* haben
- gibt *keinen Hinweis*, ob bzw. wie angewendete Aspekte die *Semantik* eines Systems geändert haben

Interessant ist auch die Fragestellung, ob ein Aspekt *wichtige Programmeigenschaften* (z.B. Deadlockfreiheit) beeinflusst.

Impact Analysis ist kein Ziel von ajdt!

Mögliche Herangehensweisen

Dynamische Analyse

- Analyse von während eines Programmlaufes gewonnenen Daten, Hintergrund: *Testen*
- **Vorteil:** Keine falschen Positive.
- **Nachteil:** Testfälle decken nie alle möglichen Programmläufe ab, *unvollständige Information*.

Statische Analyse

- Analyse des Quellcodes, Kernprinzip: *Konservative Approximation*
- **Vorteil:** Aussagen sind unabhängig von der Eingabe gültig
- **Nachteile:** konservative Approximation (\rightarrow *falsche Positive*), aufwendig

Trace Analysis (1) – Erzeugung der Traces

Ein *Trace* ist die Aufzeichnung eines Programmlaufs, z.B. aller aufgerufener Methoden.

Setting:

$$B, \mathcal{A}, \text{ Testsuite für } B : \mathcal{T} = \{T_1, \dots, T_n\}$$

Zum Erzeugen der Traces wird das Zielprogramm *instrumentiert*, bzw. das Laufzeitsystem angezapft:

- modifizierte JVM
- Java Debugging Interface
- ein Tracing-Aspekt mit AspectJ

Durchlauf von \mathcal{T} erzeugt dann die Traces.

Trace Analysis (2) – Analyse von Trace Deltas

Beispiel:

```
-> main
  -> f
    -> g
      -> h
        <- h
          <- g
            ...
<- main
```

Idee:

- Vergleich von trace_B mit trace_S (mit diff)
- Analyse der Unterschiede:
 - Primäreffekte
 - Sekundäreffekte
- Problem: Skalierbarkeit

Impact Analysis für AOP – dynamische Analyse – Trace Analysis_____

Trace Analysis (3) – Primäreffekte

Primäreffekte sind

- explizite Ausführung von advice
- durch advice geklammerte Ausführung von Base-Methoden

Primäreffekte geben direkten (auch mit ajdt) nachvollziehbaren Aspekt-Einfluss wieder.

```
-> main
  -> f
    -> before$SomeAspect$14
      <- before$SomeAspect$14
        -> around$SomeAspect$25
          -> h
            <- h
              <- around$SomeAspect$25
                ...
<- main
```

Trace Analysis (4) – Sekundäreffekte

Sekundäreffekte: Unterschied läßt sich *nicht* direkt einem Aspekt zuordnen → mögliche unerwünschte Nebenwirkung!

Mögliche Gründe:

- Geänderter Kontrollfluss:
 - Aspekt-Code wirft eine Exception
 - **around()**-advice
- Geänderter Datenfluss:
 1. Aspekt beeinflusst Systemzustand
 2. dieser führt zu geändertem Aufrufverhalten

Details: Störzer/Krinke/Breu: Trace Analysis for Aspect Application. (AAOS 2003/ECOOP 2003).

Impact Analysis für AOP – dynamische Analyse – Analyse des Callgraphen_____

Vom Trace zum dynamischen Callgraphen

Großer Nachteil von Traces: enormes Datenvolumen

→ mehrere 100 MB sind schnell erreicht

Ergebnisse zu gross: Annahme, diff reduziert auf 1%

→ mehrere MB Text pro Testlauf (Bibel: ca.4.3MB)

Automatische Auswertung? Aktuelle Forschung: Analyse, Aggregation und Klassifizierung der Trace-Deltas

Weitere Möglichkeit: Datenreduktion durch Abstraktion

→ Man geht zum (dynamischen) Callgraphen über

Abstraktion bedeutet immer auch Datenverlust:

→ hier Historie, Reihenfolge

Impact Analyse mit dynamischen Callgraphen

Der dynamische Callgraph $CG_{S_T} = (V, E)$ wird aus Traces abgeleitet: Seinen m, n Methoden (oder advice),
 $V = \{m \mid m \in S\}$. Dann gilt:

$$(m \rightarrow n) \in \text{trace}_S(T) \Rightarrow (m, n) \in E$$

Eigenschaften des Callgraphen:

- aggregierte Information, damit Informationsverlust
- deutlich kleiner \rightarrow beherrschbar!

Nachteil: Durch Datenabstraktion *Delta-Analyse nicht anwendbar*, alternatives Verfahren nötig.

Beispiel: $\text{signum}(x)$, around : $x = !x$

Impact Analysis für AOP – dynamische Analyse – Analyse des Callgraphen_____

Zusammenhang mit Qualitätssicherung

Bisher waren die *Ergebnisse der Testtreiber* irrelevant, diese werden nun mit beachtet.

Annahme: alle Testfälle sind für B erfolgreich.

Setting: $S = B \oplus \mathcal{A}$. $\mathcal{T} = \mathcal{T}_{\text{fail}} \cup \mathcal{T}_{\text{pass}}$; $\mathcal{T}_{\text{fail}} \cap \mathcal{T}_{\text{pass}} = \emptyset$.

Sichere Aspekte: $A \in \mathcal{A}$ ist ein sicherer Aspect, falls

$$A \text{ beeinflusst } T \Rightarrow T \in \mathcal{T}_{\text{pass}}.$$

Fehlersuche: Sei $T \in \mathcal{T}_{\text{fail}}$. Dann identifiziert der Callgraph alle Aspekte, die möglicherweise Ursache des Fehlers sind.

Achtung: Vorgestelltes Verfahren ist nur eine Heuristik!

Fazit Callgraphanalyse

Analyse des Callgraphen erlaubt es, die Menge

- der Aspekte, die einen Testtreiber beeinflussen und
- der Testtreiber, die von einem Aspekt beeinflusst werden

zu ermitteln. Mit den Treiber-Ergebnissen können dann *problematische Aspekte identifiziert* werden.

Die Analyse des dynamischen Callgraphen kann somit zur *Fehlersuche* eingesetzt werden.

Derzeit laufende Diplomarbeit: “Change Impact Analysis for AOP using dynamic Callgraphs”

Impact Analysis für AOP – dynamische Analyse – Analyse des Callgraphen_____

Probleme und offene Fragen

Dynamische Analyse ist ähnlich wie Testen: Fehler können Nachgewiesen werden, aber *nicht deren Abwesenheit!*

- Zuverlässigkeit der Ergebnisse?
 - Keine Beeinflussung → Testsuite zu schlecht?
 - Zusammenhang mit Coverage der Testsuite?
- Dynamische Callgraphen bieten nur unvollständige Information:
 - Weitere Anwendbarkeit dynamischer Callgraphen?
 - Vergleich mit statischen Callgraphen?

Statische Analyse: Ein statischer AO Callgraph

Betrachtete dynamische Analysen können bei Programmierung und Fehlersuche unterstützen, gestatten aber *keine allgemeingültigen Aussagen*.

→ statische Analyse erforderlich.

Beobachtung: Callgraph-Analysen meist von der Art der unterliegenden Datenstruktur (statisch/dynamisch) unabhängig. Konsequenz:

- dynamische Variante gut für Prototyping, Programmierersupport
- statische Variante nötig für semantische Garantien

Impact Analysis für AOP – statische Analyse – Analyse des Callgraphen_____

Impact Analyse mit statischen Callgraphen

Wir betrachten nun *statische* Callgraphen. Berechnung:

Basis Sourcecode: Mögliche Calls müssen aus dem Quellcode ermittelt werden, dies impliziert

1. Parsen des Codes
2. Namensauflösung und Typisierung von Ausdrücken
3. Auflösen von Overloading, Approximieren der dynamischen Bindung: Typanalyse (RTA/XTA, PointsTo)!
4. Advice: Approximieren dynamischer Pointcuts (cflow, if), Auflösen von Wrappinghierarchien

Prinzip der konservativen Approximation: Alle theoretisch möglichen Aufrufe müssen erfasst werden

Folge: falsche Positive.

Warum ist konservative Approximation nötig?

Exakte PointsTo Analyse für OO-Sprachen ist *unentscheidbar*.

Beispiel:

```
class A          { void n() {} }
class B extends A { void n() {} }
class C {
    psv main(String[] args) {
        A a = null;
        if (args.length > 1) a = new A();
        else                   a = new B();
        a.n();
    }
}
```

Welche Methode wird tatsächlich aufgerufen?

Impact Analysis für AOP – statische Analyse – Analyse des Callgraphen_____

Auflösen von Wrappinghierarchien

Nur als Teaser ...

Definition 1 (ζ : Advice execution per joinpoint) Let $\mu = \text{head}(\xi_{jp})$ and $\nu = \rho(\text{head}(\xi_{jp}))$ for around-advice. Let

$\zeta : \mathcal{ADV} \times (V \times V) \times V \rightarrow (V \times V) \times V$ be defined as follows:

$$\zeta(\xi_{jp}, E, i) = \begin{cases} (E \cup \bigcup_{j \in \text{target}(jp)} (i, j), i), & \text{if } \xi_{jp} = [] \wedge \text{kindof}(jp) = \text{call} \\ (E, i), & \text{if } \xi_{jp} = [] \wedge \text{kindof}(jp) \neq \text{call} \\ \zeta(\text{tail}(\xi_{jp}), E \cup (i, \mu) \cup (\mu, \nu), \nu), & \text{if } \xi_{jp} \neq [] \wedge \text{kindof}(\mu) = \text{around} \\ \zeta(\text{tail}(\xi_{jp}), E \cup (i, \mu), i), & \text{if } \xi_{jp} \neq [] \wedge \text{kindof}(\mu) \in \{\text{before}, \text{aft}\} \end{cases}$$

Definition 2 (ς : Called items per method) Let $\varsigma : (V \times (V \times V)) \times \mathbb{J} \rightarrow V \times (V \times V)$ be defined as

$$\varsigma((i, E), \sigma_i) = \begin{cases} (i, E), & \text{if } \sigma_i = [], \\ \varsigma(\zeta(\xi(\text{head}(\sigma_i)), E, i), \text{tail}(\sigma_i)), & \text{if } \text{kindof}(\text{head}(\sigma_i)) = \text{exec}(i) \\ \varsigma(i, \pi_E(\zeta(\xi(\text{head}(\sigma_i)), i)), \text{tail}(\sigma_i)), & \text{if } \text{kindof}(\text{head}(\sigma_i)) \neq \text{exec}(i) \end{cases}$$

Anwendung: Prüfung von Aspekt Kommutativität

$S = B \oplus \mathcal{A}$ mit $\mathcal{A} = \{A_1, A_2\}$.

Frage: Gilt $B \oplus A_1 \oplus A_2 = B \oplus A_2 \oplus A_1$?

→ Im Allgemeinen natürlich nicht.

Warum ist diese Frage aber entscheidend für die Semantik eines Systems?

- Aspekte sollen in *grossen* Systemen eingesetzt werden
→ verteilte Arbeit, ggf. unterschiedliche Teams
- Unabhängig voneinander entwickelte Aspekte beeinflussen selbe Stelle im Programm → **potentieller Konflikt!**
- Wer ist verantwortlich? Team B , Team A_1 , Team A_2 ?

Compiler wählt *zufällige Ordnung* (AspectJ: alphabetisch).

Impact Analysis für AOP – statische Analyse – kommutative Aspekte

Prüfung “Composition Soundness”

Definition 3 (a_1 liest von a_2) Sei a ein advice, $\text{write}(a)$ die Menge der von a geschriebenen, $\text{read}(a)$ die Menge der von a gelesenen Datenobjekte.

Es gilt: a_1 liest von $a_2 \Leftrightarrow \text{commonJP}(a_1, a_2) \wedge$

$$\text{write}(a_1) \cap \text{write}(a_2) \neq \emptyset \vee \text{read}(a_1) \cap \text{write}(a_2) \neq \emptyset.$$

In Anlehnung an die Transaktionstheorie (Datenbanken):

A_1, A_2 Aspekte, $\text{conflict}(A_1, A_2) \Leftrightarrow$

$$\exists a_1 \in A_1, a_2 \in A_2 : a_1 \text{ liest von } a_2$$

oder umgekehrt.

Prüfung “Composition Soundness” (2)

Was hat das mit dem Callgraphen zu tun?

Nötig: Berechnung $\text{read}(a), \text{write}(a)$

→ erfordert transitive Betrachtung aller von a aufgerufener Methoden – das kann am Callgraphen abgelesen werden.

Falls $\text{conflict}(A_1, A_2)$, muss der Programmierer *explizit eine Reihenfolge angeben*, also entweder $B \oplus A_1 \oplus A_2$ oder $B \oplus A_2 \oplus A_1$.

Verantwortlich: Team A_1 und Team A_2 .

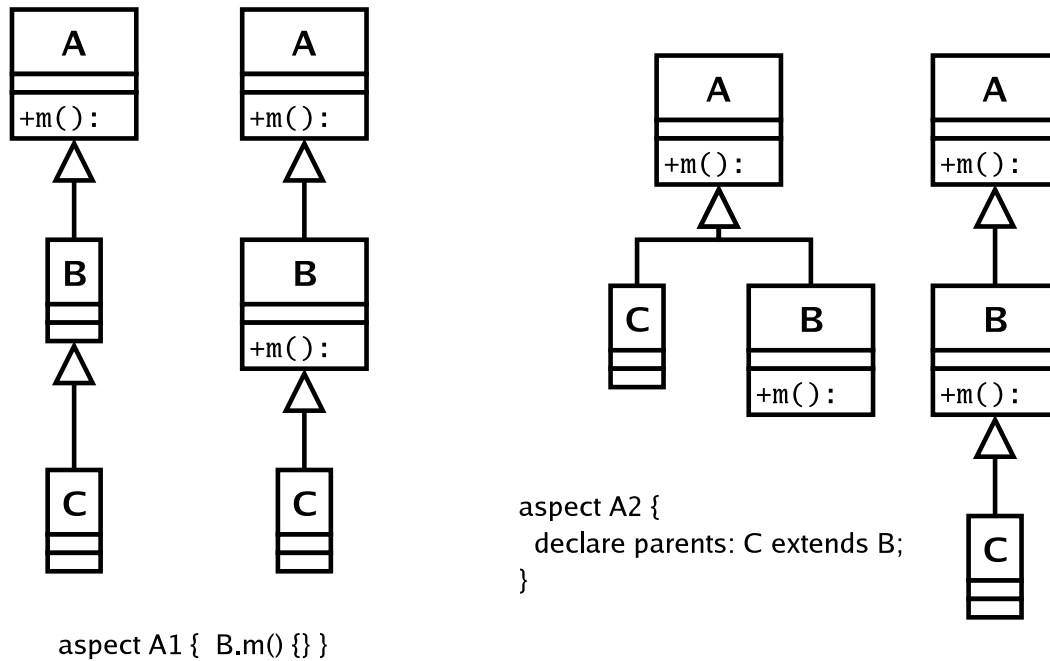
Details: M. Stözer, “Checking Aspect Commutativity using the Callgraph”, submitted for publication.

Impact Analysis für AOP – statische Analyse – ITDs und Changed Lookups_____

System-Semantik und Inter Type Declarations

- Können ITDs die Semantik eines Systems ändern?
- Ja! Überschreiben von in Oberklassen definierten Methoden
- Ja! Ändern der Vererbungshierarchie; neue Oberklasse kann ebenfalls Methoden der alten Oberklasse überschreiben.

Semantik Änderung durch ITDs – Beispiele



Betrachte jeweils den Aufruf `C.m()` ;.

Impact Analysis für AOP – statische Analyse – ITDs und Changed Lookups_____

Analyse von ITDs

Sei `staticLookup` eine Funktion, die für jede deklarierte Methode die definierende Klasse liefert (d.h. die “nächste” Klasse, die diese Methode implementiert).

1. **Idee:** ermittle `staticLookup` mit und ohne ITDs
2. Semantikerhaltung nur, falls
 $\text{staticLookup}_{\text{old}} = \text{staticLookup}_{\text{new}}$.

Erhöhung der Präzision:

- Einschränkung auf verwendete Typen (RTA/XTA, PointsTo)
- Integration in CallGraphAnalyse

Kurze Zusammenfassung

- AOP und AspectJ, Idee und Wirkungsweise
- Probleme bei derzeitigen AO-Sprachen: fragile Pointcuts, Auswirkungen: kaum semantischer Support
- Impact Analyse
 - Trace Delta Analysis
 - dynamische Callgraphen als Heuristik/zur Fehlersuche
 - Aspekt Kommutativität als Anwendung für statischen Callgraphen
 - Analyse von ITDs

Roundup

Fazit

AOP ist eine sehr interessante neue Entwicklung im Bereich Software Engineering, derzeit auch Forschungs-Modethema

- Für Development Aspekte heute schon vorbehaltlos zu empfehlen.
- Für Production Aspekte ist der semantische Background noch unbefriedigend – daran wird aber gearbeitet!

Überbegriff: Aspect Oriented Software Development (AOSD), umfasst alle Bereiche des Software-Lebenszyklus (Requirements-Engineering bis Wartung).

Was ich nicht erwähnt habe ...

Beweise: Die Korrektheit einiger Verfahren ist bewiesen.

Technische Details: hier steckt bekanntlich der Teufel ...

** WERBUNG **

Im Bereich Analyse von AO-Programmen sind laufend Themen für *Programmierpraktika, Bachelor- und Diplomarbeiten* zu vergeben.

Fragen?

Fragen?