© Markus Knauß, 2007

Risiken und Nebenwirkungen der AOP

Markus Knauß
Abteilung Software Engineering, Institut für Softwaretechnologie
Universität Stuttgart
knauss@informatik.uni-stuttgart.de

AOP-Day 07, 19. April 2007

© Markus Knauß, 2007

AOP – Ja oder Nein?

AOP verspricht eine bessere Modularisierung und dadurch eine bessere Wartbarkeit von Software. Darum sollte AOP in einem neuen Projekt eingesetzt werden.

Da AOP schon 10 Jahre alt ist und Gartner 2004 die Spitze für AOP für 2006-2011 sah, sollten die Versprechungen an existierenden großen AOP Projekten geprüft werden.

© Markus Knauß, 2007

AOP – Ja oder Nein?

Es konnten keine praxisrelevanten Referenzprojekte gefunden werden, die für die Entscheidung pro oder contra AOP geeignet gewesen wären.

Verfügbar sind:

- Forschungsprojekte
- Lehrbücher, Pro- und Contra-Schriften
- kleine Beispiele: Tracing, Logging, GoF-Patterns

Darum wurden die Auswirkungen der AOP auf die Wartbarkeit theoretisch untersucht. Die Ergebnisse stellt dieser Vortrag vor.

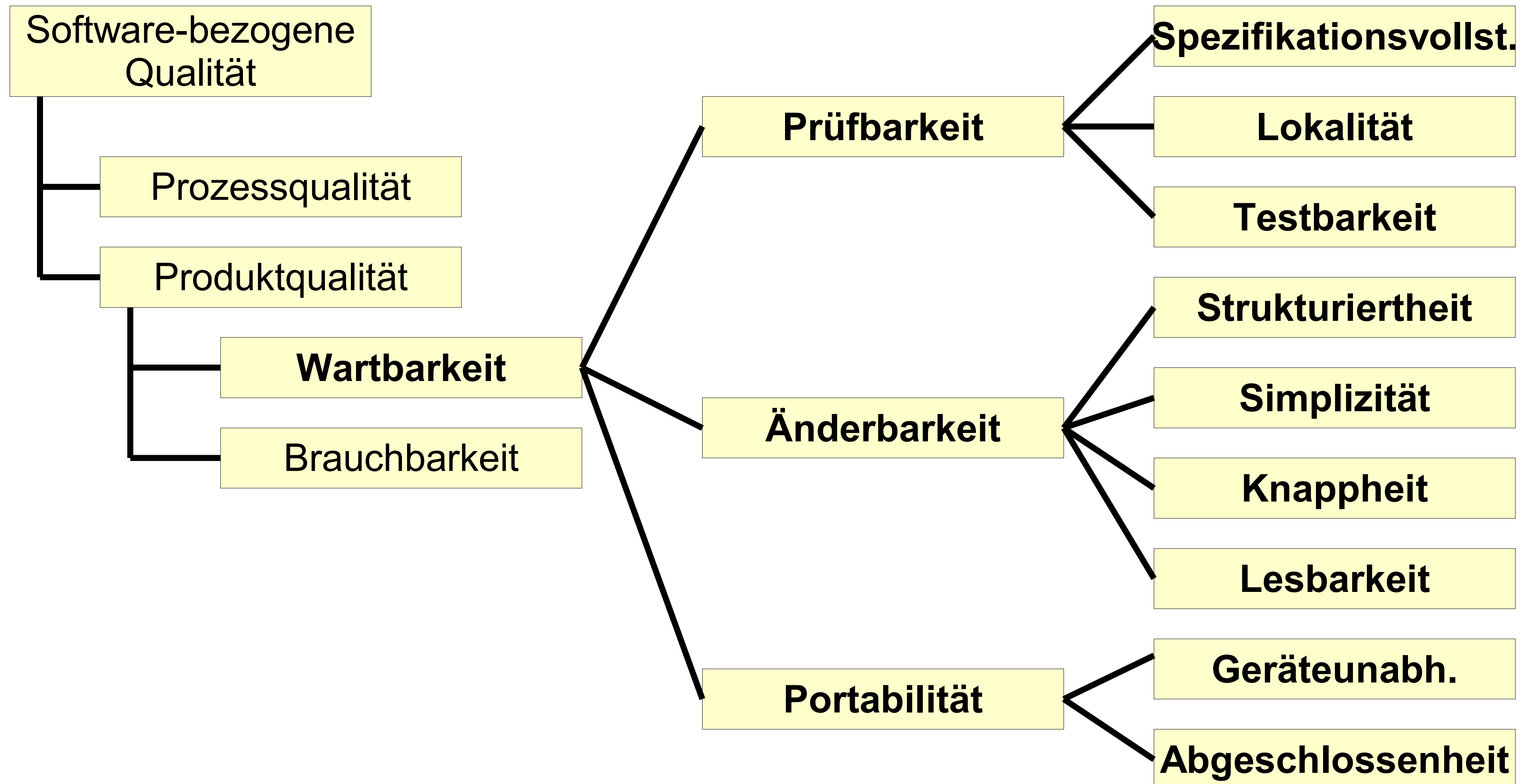
© Markus Knauß, 2007

Inhalt

- Wartbarkeit
- Auswirkungen der AOP auf die Wartbarkeit
- Zusammenfassung und Empfehlung

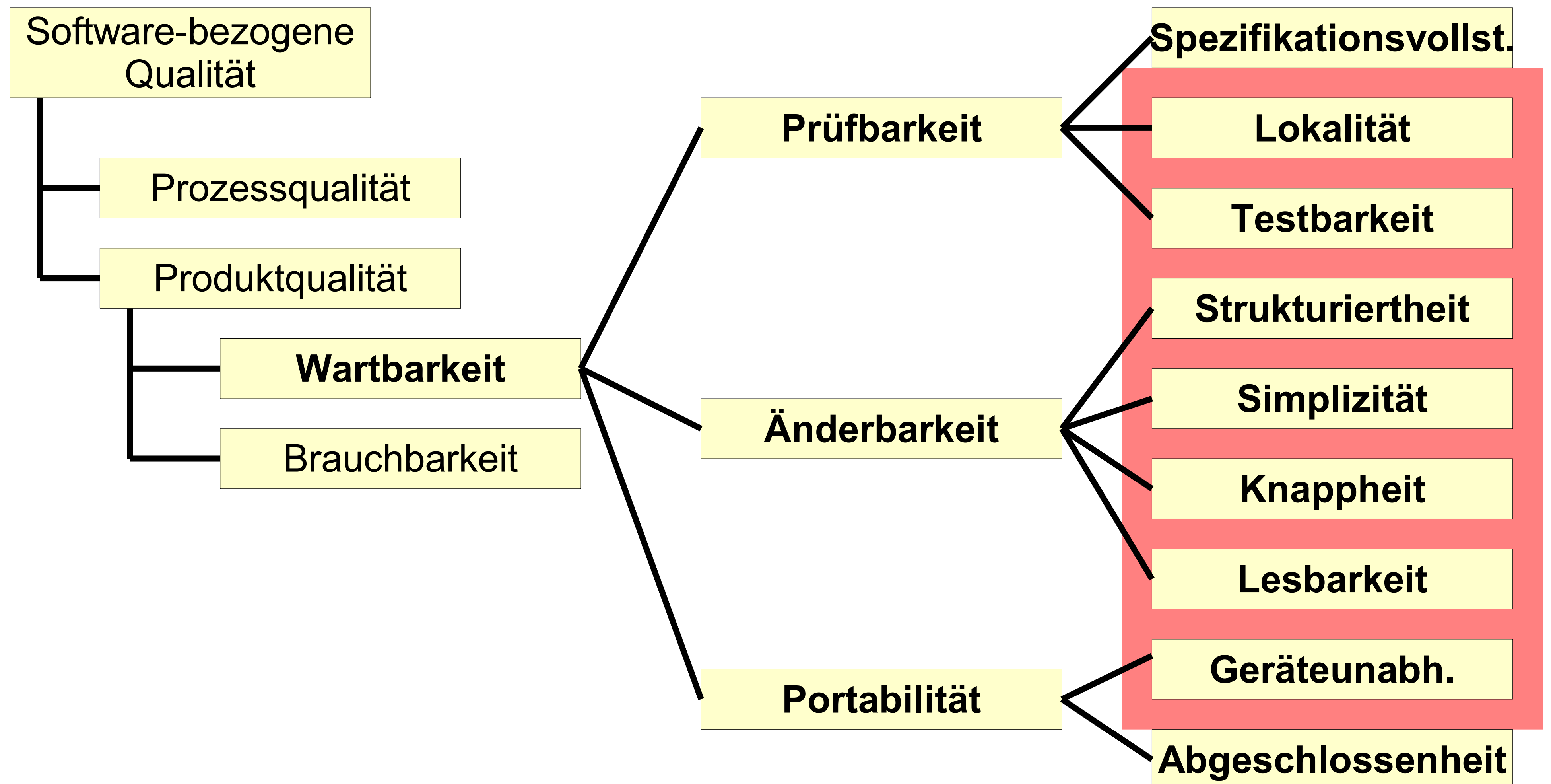
© Markus Knauß, 2007

Wartbarkeit



© Markus Knauß, 2007

Auswirkungen der AOP auf die Wartbarkeit



© Markus Knauß, 2007

Lokalität

„Die Lokalität einer Software ist hoch, wenn Fernwirkungen in der Software (Wirkungen über die Grenzen einer Software-Komponente hinweg) vermieden sind.“ (Ludewig und Lichter, 2007)

Die AOP ermöglicht unsichtbare Fernwirkungen und verschlechtert hierdurch die Lokalität.

Advice kann an beliebigen Stellen in den Programmcode eingefügt werden. Der *Advice* kann durch Änderung des Programmzustands oder des Kontrollflusses das Verhalten des Programms verändern. Ebenso kann eine Änderung des Programmcodes das Verhalten des *Advice* verändern.

→ Darum wird die Lokalität durch die AOP verschlechtert.

© Markus Knauß, 2007

Testbarkeit

„Die Testbarkeit einer Software ist hoch, wenn die Programme unter definierten Bedingungen ausgeführt und die relevanten Resultate vollständig erfasst werden können.“ (→ reproduzierbar) (Ludewig und Lichter, 2007)

Der Test von Programmteilen ist erst nach der *Composition* der *Concerns* möglich. Ist die *Composition* nicht deterministisch, dann ist ein reproduzierbarer Systemtest unmöglich.

→ Die notwendige *Composition* getrennt implementierter *Crosscutting Concerns* verschlechtert die Testbarkeit und kann einen reproduzierbaren Test sogar unmöglich machen.

© Markus Knauß, 2007

Strukturiertheit

„Die Strukturiertheit ist hoch, wenn die Software in logisch abgeschlossenen Einheiten mit hohem Zusammenhalt und geringer Kopplung gegliedert ist.“ (Ludewig und Lichter, 2007)

Die konsequente Trennung von *crosscutting Concerns* führt zu einem hohen Zusammenhalt wohingegen die Kopplung auf die schlechteste Stufe – den Einbruch – abgewertet wird.

Advice kann an beliebigen Stellen Programmcode verändern.

→ Die Veränderung entspricht einem Einbruch in den Programmcode. Darum ist die Kopplung auf der schlechtesten Stufe.

© Markus Knauß, 2007

Simplizität

„Die Simplizität einer Software ist hoch, wenn in der Software nur wenige schwer verständliche Konstruktionen enthalten sind.“ (Ludewig und Lichter, 2007)

Ein Programm ist simpel (einfach, schlicht), wenn es leicht verstanden werden kann.

Auf den ersten Blick vereinfacht die getrennte Implementierung der *crosscutting Concerns* das Verstehen. Bei näherer Betrachtung stellt man fest, dass das Programm sehr viel schwerer zu verstehen ist, da die *Composition* im Kopf ausgeführt werden muss. Erst nach der *Composition* kann man verstehen, was das Programm wirklich tut.

→ Darum entstehen durch die AOP schwer verständliche nicht simple Programme.

© Markus Knauß, 2007

Knappheit

„Die Knappheit der Software ist hoch, wenn ihr Umfang durch Vermeidung von Redundanz aller Art gering gehalten wurde.“ (Ludewig und Lichter, 2007)

Die getrennte Implementierung der *crosscutting Concerns* führt zu knapperen Programmen.

Da anstatt vieler gleichförmiger Methodenaufrufe nur noch ein *Pointcut Designator* geschrieben werden muss, sind die Programme wesentlich knapper.

© Markus Knauß, 2007

Lesbarkeit

„Die Lesbarkeit einer Software ist hoch, wenn ein (fremder) Leser in der Lage ist, mit minimalem Aufwand den Inhalt korrekt zu erfassen.“ (Ludewig und Lichter, 2007)

Obliviousness führt zu einer schlechteren Lesbarkeit.

Ein (fremder) Leser muss zunächst alle Stellen erkennen, an denen *Advice* stattfindet, und muss den *Advice* korrekt einfügen. Erst dann hat er eine Chance, den Inhalt eines Programms korrekt zu erfassen. Der Aufwand für das korrekte Erfassen des Inhalts ist mit *Advice* wesentlicher schwerer als ohne.

→ AOP führt zu einer Verschlechterung der Lesbarkeit.

© Markus Knauß, 2007

Geräteunabhängigkeit

„Die Geräteunabhängigkeit einer Software ist hoch, wenn Merkmale spezifischer Geräte darin eine Rolle spielen.“ (Ludewig und Lichter, 2007)

AOP stellt keine Anforderungen an die Hardware. Darum sind Geräte in dieser Betrachtung spezielle Softwarewerkzeuge und Laufzeitumgebungen.

Aspektorientierte Programme führen zur Abhängigkeit von speziellen Übersetzern, den *Weavern*, und (eventuell) speziellen Laufzeitumgebungen.

→ AOP verschlechtert die Geräteunabhängigkeit

© Markus Knauß, 2007

Zusammenfassung

Positive Auswirkungen
- Knappheit

Negative Auswirkungen
- Lokalität
- Testbarkeit
- Strukturiertheit
- Simplizität
- Lesbarkeit
- Geräteunabhängigkeit

Fazit: Die AOP verschlechtert die Wartbarkeit drastisch.

© Markus Knauß, 2007

Zusammenfassung

Obliviousness und ***Quantification***, zwei Schlüsselwörter der AOP, verschlechtern die Wartbarkeit drastisch.

Obliviousness führt dazu, dass weder der Entwickler noch der Compiler sehen kann was tatsächlich passiert.

Quantification bewirkt, dass an jeder Stelle in einem Programm beliebiger Unsinn getrieben werden kann.

→ Einfache Änderungen werden zu einem Glücksspiel mit geringen Gewinnchancen.

Erschwerend kommt hinzu, dass die AOP die **Kopplung** bis hin zur **Stufe des Einbruchs** verschlechtert. Dies spricht gegen jegliche Entwurfserfahrung, die seit Dijkstra (1968) und Parnas (1972) gesammelt wurde.

© Markus Knauß, 2007

Empfehlung

Wenn Aspekte verwendet werden, dann sollte berücksichtigt werden, dass die **Kopplung** nicht verschlechtert wird und das Systemverhalten deterministisch ist. Das bedeutet:

- Keine Veränderung eines Typs, des Programmzustands oder des Kontrollflusses durch *Advice*. Aspekte müssen frei von Seiteneffekten sein.

Außer Logging und Tracing bleibt dann nicht mehr viel. Das könnte aber auch ohne große Abhängigkeit von Werkzeugen implementiert werden, zum Beispiel durch einen Methodenaufruf.

© Markus Knauß, 2007

Inhalt

- Wartbarkeit
- Auswirkungen der AOP auf die Wartbarkeit
- Zusammenfassung und Empfehlung

© Markus Knauß, 2007

© Markus Knauß, 2007

Anhang

© Markus Knauß, 2007

Kopplung und Zusammenhalt

Beim Entwerfen ist man bemüht, eine Software zu konzipieren, deren Module einen **hohen Zusammenhalt** und eine **geringe Kopplung** haben.

Beispiel: Die idealen Knödel (schwäbisch Knöpfle) haben einen hohen Zusammenhalt und keine Kopplung. Sie zerfallen nicht (Zusammenhalt) und kleben nicht aneinander (Kopplung).

Kopplung und Zusammenhalt können in **Stufen** bewertet werden (siehe: Ludewig und Lichter, 2007).

© Markus Knauß, 2007

Zusammenhalt

Stufe	innerhalb einer Prozedur/eines Moduls	erreichbar für
kein Zusammenhalt	Die Zusammenstellung ist zufällig	Module
Ähnlichkeit	Die Teile dienen z. B. einem ähnlichen Zweck	Module
zeitliche Nähe	Die Teile werden zur selben Zeit ausgeführt	Module, Prozeduren
gemeinsame Daten	Die Teile sind durch gemeinsame Daten verbunden	Module, Prozeduren
Hersteller / Verbraucher	Der eine Teil erzeugt, was der andere verbraucht	Module, Prozeduren
einziges Datum (Kapselung)	Die Teile realisieren alle Operationen, die auf eine gekapselte Datenstruktur möglich sind	Module, Prozeduren
einzige Operation	Operation, die nicht mehr zerlegbar ist	Prozeduren

© Markus Knauß, 2007

Kopplung

Stufe	zwischen Prozeduren/Modulen	erreichbar für
Einbruch	Der Code kann verändert werden	
volle Öffnung	Auf alle Daten, z. B. auf globale Daten in einem C-Programm, kann zugegriffen werden.	(Module), Prozeduren
selektive Öffnung	Bestimmte Variablen sind zugänglich, global oder durch expliziten Export und Import.	Module, Prozeduren
Prozedurkopplung	Die Prozeduren verschiedener Module sind nur durch Parameter oder Funktionen gekoppelt.	Module, (Prozeduren)
Funktionskopplung	Die Prozeduren verschiedener Module sind nur durch Wertparameter und Funktionsresultate gekoppelt.	Module, (Prozeduren)
keine Kopplung	Es besteht keine Beziehung zwischen den Modulen, der Zugriff ist syntaktisch unmöglich.	Module

© Markus Knauß, 2007

Programmverstehen

Um ein Programm zu verstehen, konstruiert sich ein Entwickler eine Vorstellung der Funktion und des Aufbaus des Programms in seinem Kopf; ein mentales Modell. Das mentale Modell wird Top-Down und Bottom-Up erzeugt. Im wesentlichen geht es darum, Struktur und Funktion im Programm zu erkennen und die eigene Vorstellung über Funktion und Struktur mit der tatsächlichen des Programms in Einklang zu bringen (Mayrhauser und Vans, 1995).

Die Erzeugung des mentalen Modells wird durch Fragen vorangetrieben, die Entwickler an das Programm stellen (Erdös und Sneed, 1998; Sillito et al. 2006).

Je einfacher die Fragen zu beantworten sind, desto schneller kann des mentale Modell erstellt werden und das bedeutet, dass das Programm einfacher zu verstehen ist.

© Markus Knauß, 2007

Literatur

Dijkstra, E. W. (1968): The structure of THE multiprogramming system. Communications of the ACM, 11(5), 341-346.

Erdös, Katalin und Harry M. Sneed (1998): Partial Comprehension of Complex Programs (enough to perform maintenance). in Proceedings of the 6th Intl. Workshop on Program Comprehension, 98-105.

Filman, Robert E., Tzilla Elrad, Sobhán Clarke und Mehmet Akşit (2005):Aspect-Oriented Software Development. Pearson Education Inc.

Ludewig, Jochen und Horst Lichter (2007): Software Engineering – Grundlagen, Menschen, Prozesse, Techniken. dpunkt.verlag GmbH.

Mayrhauser, Anneliese von und A. Marie Vans (1995): Program Comprehension During Software Maintenance and Evolution. IEEE Computer, 28(8), 44-55.

Parnas, D. L. (1972): On the criteria to be used in decomposing systems into modules. Communications of the ACM. 15(12), 1053-1058.

Sillito, Jonathan, Gail C. Murphy und Kris De Volder (2006): Questions Programmers Ask During Software Evolution Tasks. in Proceedings of the 14th ACM SIGSOFT Intl. Symp. on Foundations of Software Engineering, 23-34.