Forschungsmethodik in der Softwaretechnik

Wie schreibe ich eine Dissertation in der Software-Forschung?

Walter F. Tichy

IPD Tichy, Fakultät für Informatik

Inhalt

- Teil I: Ratschläge für eine schlechte Dissertation
  - Mit schamlosen Anleihen (4 Folien) bei David Patterson: „How to have a Bad Career in Research/Academia“ http://www.cs.berkeley.edu/~pattrsn/talks/BadCareer.pdf
- Teil II: Wie weiß ich, dass ich einen Fortschritt erreicht habe?
  - Experiment
  - Analyse von Software-Depots
  - Benchmark

Teil I:
Ratschläge für eine schlechte Dissertation

Schlechte Entscheidung #1: Mach es kompliziert!

- Bestes Kompliment für einen Forscher:
  „Es ist so kompliziert, dass ich die Idee nicht verstehe.“
- Komplexität macht´s einfacher für weitere Beiträge:
  Wenn´s niemand versteht, wer kann einem dann widersprechen?
- Von was Kompliziertem fällt es leichter, kleine Inkremente zu publizieren.
- Falls etwas einfach wäre, wie könnten andere Wissenschaftler deine Brillanz, Intellekt, Tiefgang und Detailwissen erkennen und bewundern?

Quelle: David Patterson

Schlechte Entscheidung #2: Lass dich nie widerlegen!

Quelle: David Patterson

- Vermeide Erfolgskriterien, Implementierung
- Vermeide quantitative Experimente
  - Bei guter Intuition braucht man keine Versuche!
  - Warum sollte man Kritikern Munition liefern?
  - Versuche dauern zu lange. Lieber mehr Ideen generieren.
  - Niemand erwartet quantitative Ergebnisse.
- Vermeide Benchmarks
  - Warum mit anderen vergleichen? Wir wissen, was richtig ist.
  - Wenn es ein verrückter Student doch implementiert und testet, dann sag:
    - Ist nicht für deine Anwendung/Kontext
    - Wird erst in 20 Jahren wichtig (gibt dir 19 sichere Jahre)
  - Benchmarks führen in die falsche Richtung. Jede Menge super Ideen werden vernichtet, nur weil sie jemand ausprobiert.

Schlechte Entscheidung #3: Benutze die Software-Wissenschaftsmethode!

Überholte wiss. Methode

- Hypothese
- Folge von Experimenten
- ändere 1 Parameter/Experiment
- Bestätige/widerlege Hypothese
- Dokumentiere, damit andere replizieren können.

Software-Wissenschaftsmethode

- Ahnung
- 1 Fallstudie
- ändere alle Parameter auf einmal
- Wegwerfen, falls Ahnung nicht bestätigt
- Warum Zeit verschwenden? Wir kennen die Ergebnisse
- Können mehr Ideen generieren, wenn niemand reproduzieren muss.

Quelle: David Patterson

Schlechte Entscheidung #4;
Lass Dich nicht ablenken! (vermeide Rückkopplung!)

- Sprich nicht mit Kollegen. Deren Probleme zu diskutieren kosten dich nur Zeit.
- Lies nur, was Dein Chef vorschlägt (wenn überhaupt).
- Verliere keine Zeit und Geld mit Konferenzen.
- Verschwende keine Zeit mit dem Polieren von Veröffentlichungen oder Vorträgen (missionarischer Eifer).
- Lass Dich nicht durch Nutzer oder Industrie verderben.
- Misstraue Deinem Doktorvater. Der hat nur seine eigene Karriere im Sinn.
- Arbeite nur die Stunden, für die du bezahlt wirst.
  Lass dich nicht von der herrschenden Klasse ausnützen.
- (aber Studenten muss man richtig ran nehmen.)

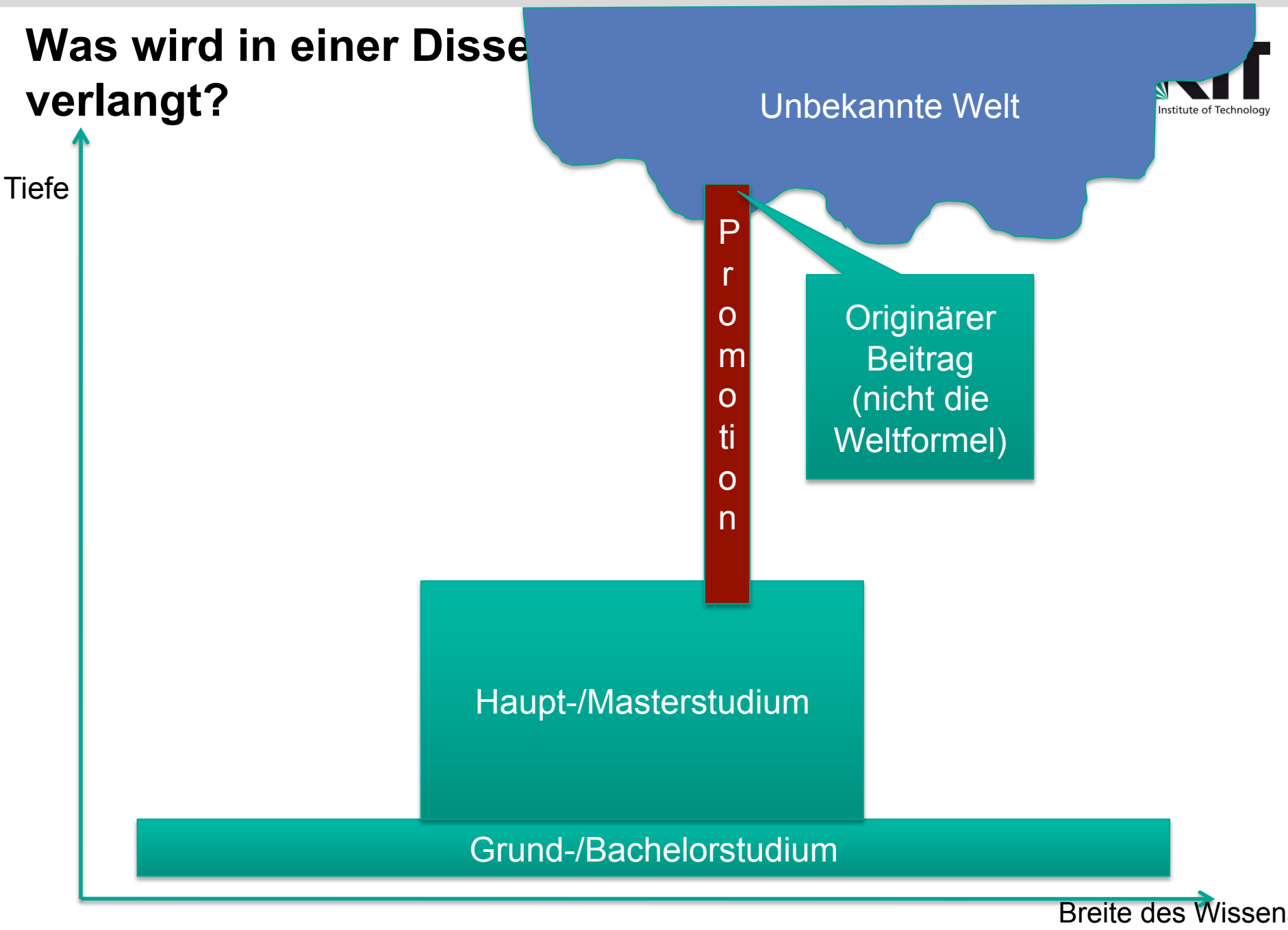
Quelle: David Patterson

Und was wären gute Entscheidungen?

- Natürlich das genaue Gegenteil!
- Für Begründung, siehe David Patterson, „How to have a Bad Career in Research/Academia“ http://www.cs.berkeley.edu/~pattrsn/talks/BadCareer.pdf
- Auch als Video erhältlich
- Sehr empfehlenswert!

Teil II:
Wie weiß ich, dass ich einen Fortschritt erzielt habe?

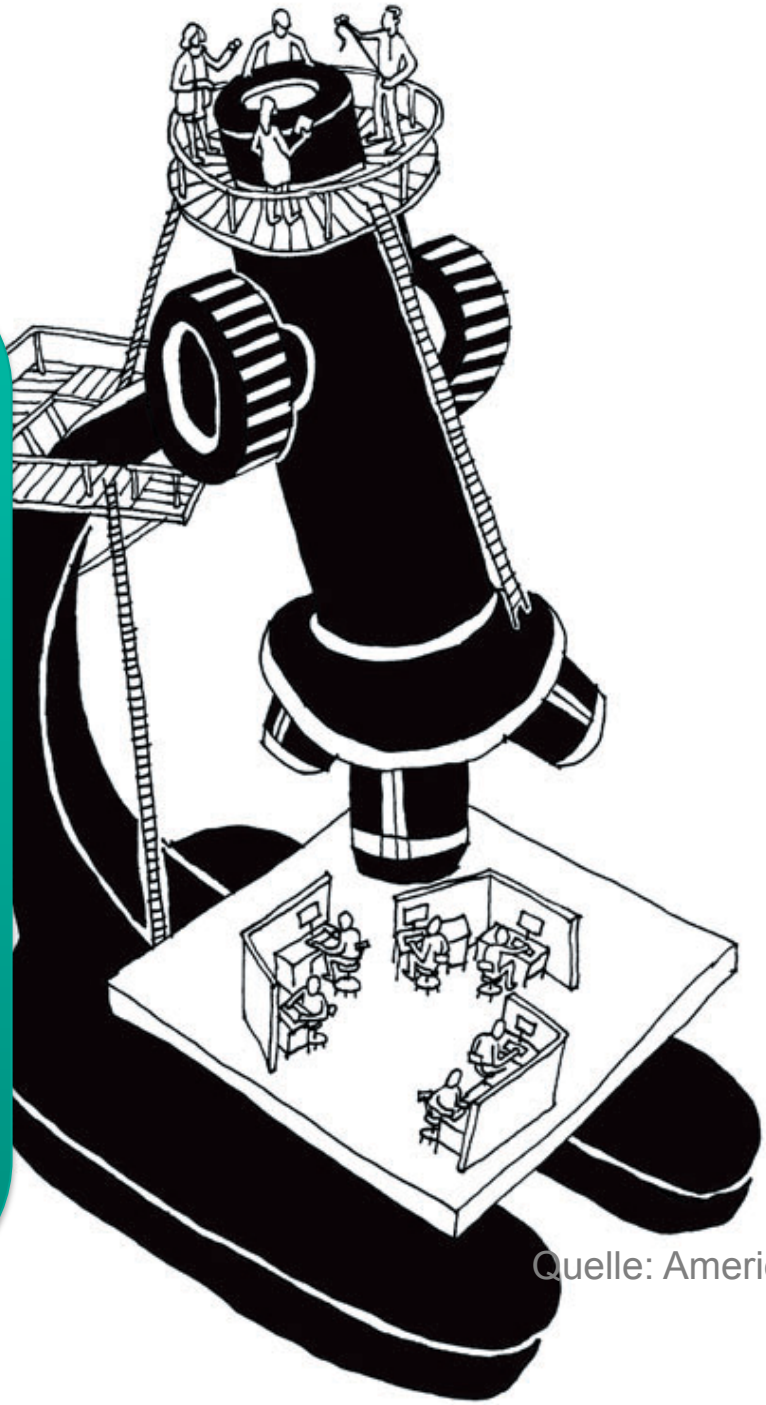
Was wird in einer Disse
verlangt?



Softwareforscher bei der Arbeit

Ewig gleiche Grundfragen:

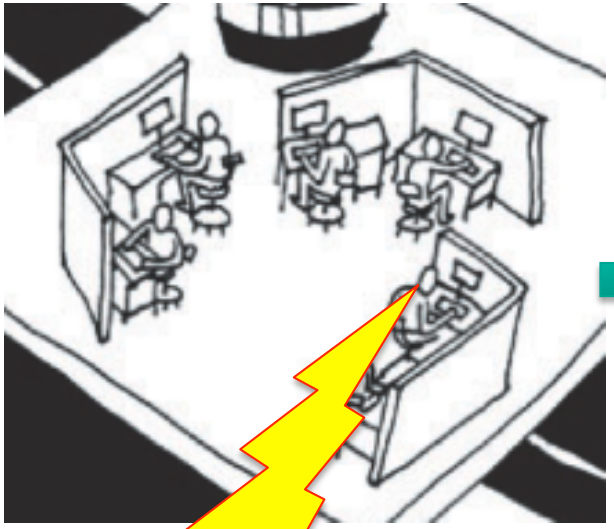
1. Wie kann man SW besser konstruieren (schneller, günstiger)?
2. Wie kann man bessere Software konstruieren (zuverlässiger, sicherer, brauchbarer, etc.)?
3. Und wie zeigt man, dass man 1. oder 2. erreicht hat?



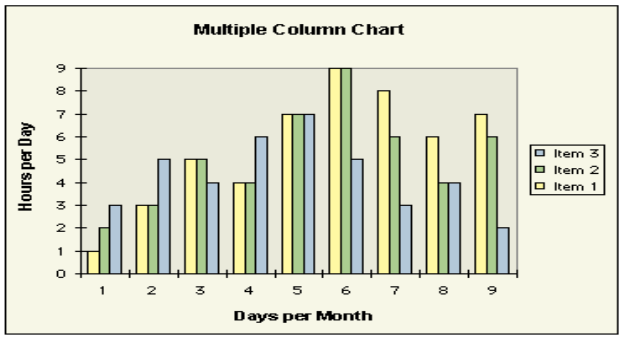
Quelle: American Scientist 6/2006

Das kontrollierte, randomisierte Experiment

variiere
unabhängige
Variablen



kontrolliere
Störvariablen



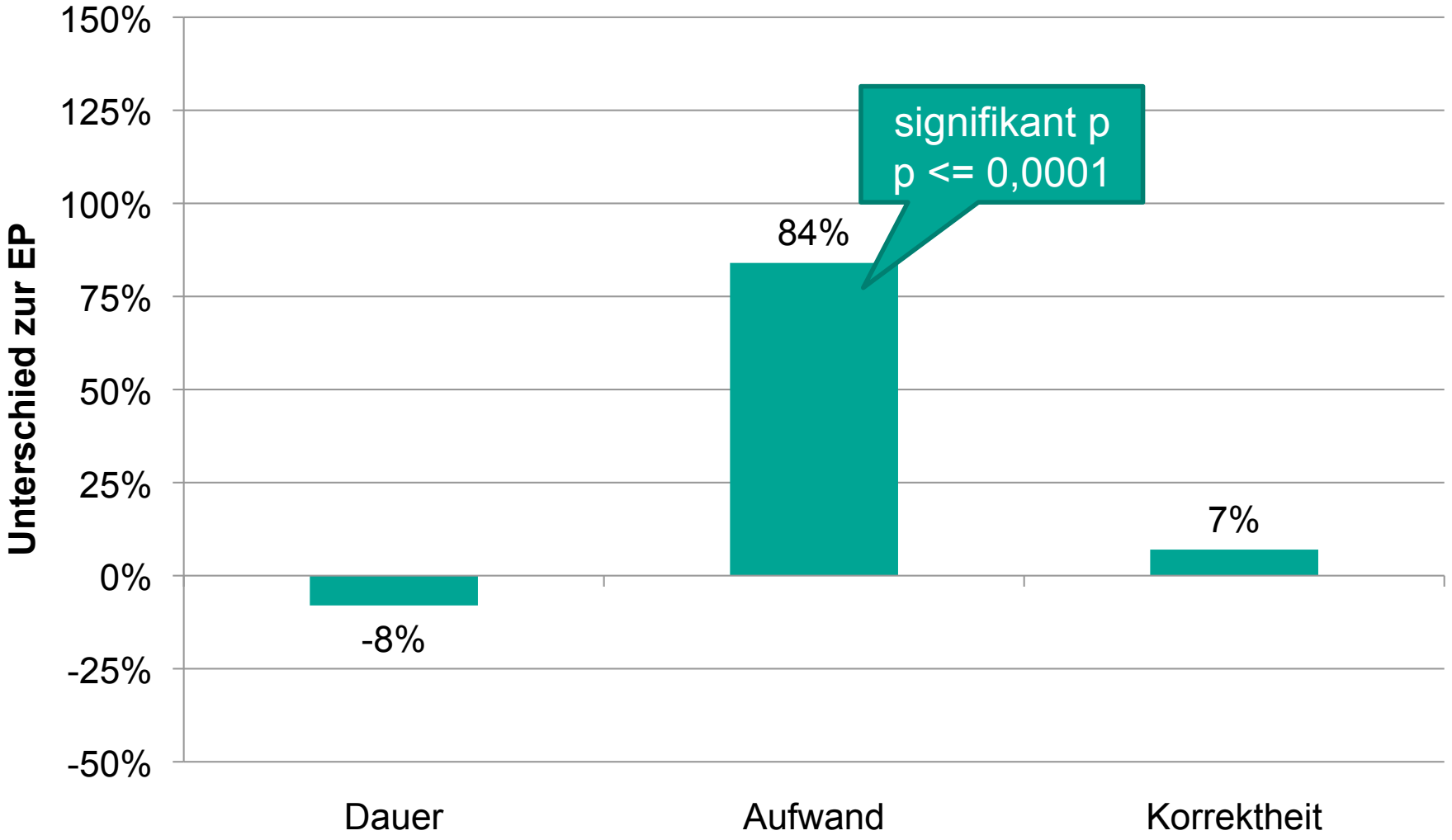
Beobachte
abhängige
Variablen

Beispiel: Experiment zum Paarprogrammieren

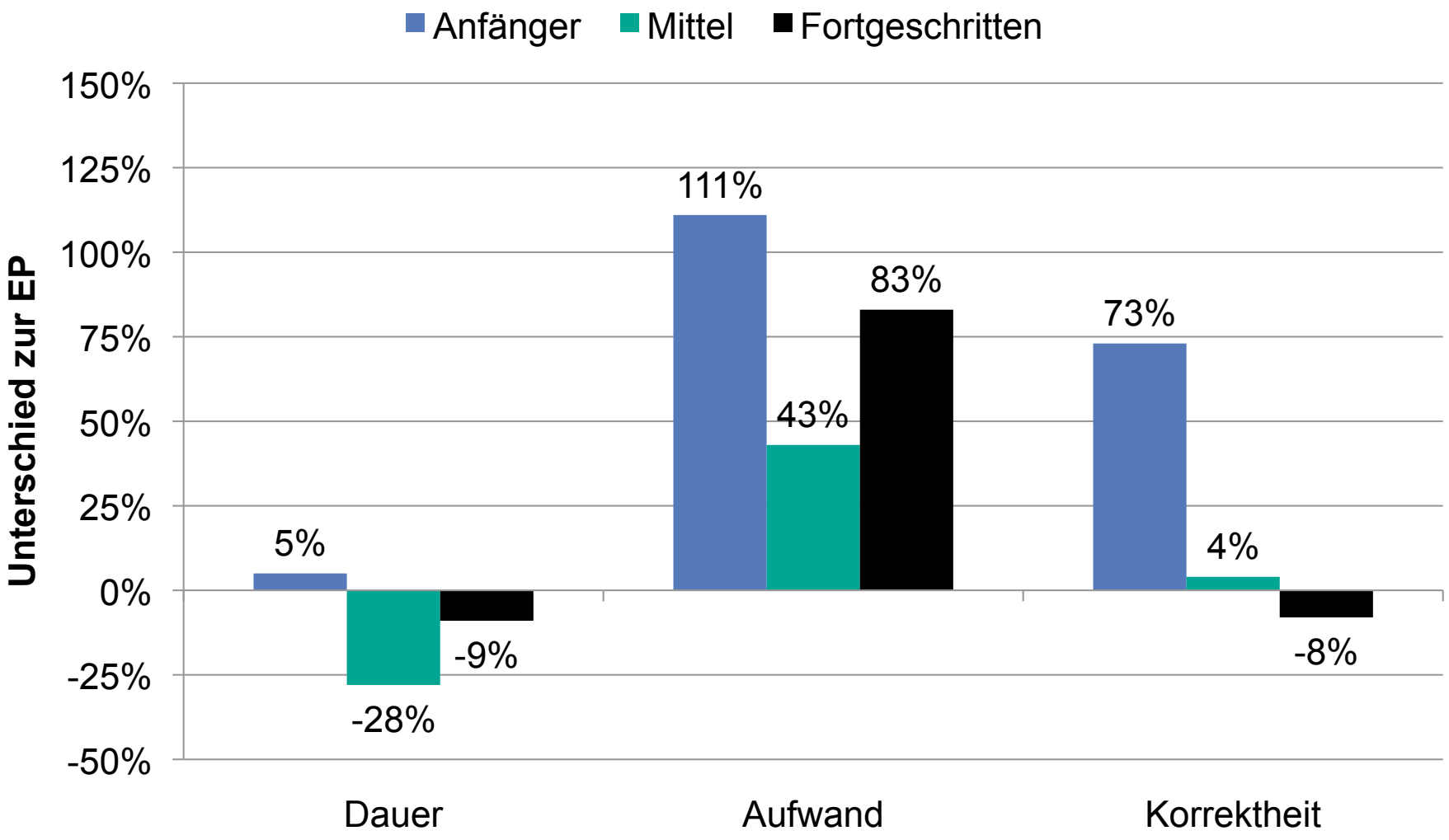
- 295 professionelle Berater (!)
- Aufgeteilt in 99 Einzelprogrammierer und 98 Paare
- aus 29 Beraterfirmen in Norwegen, Schweden und GB:
  - Accenture
  - Cap Gemini
  - Oracle
  - u. a.
- Teilnehmer wurden für fünf Stunden bezahlt.
- Kosten dafür: ca. € 250.000

Erik Arisholm, Hans Gallis, Tore Dyba, Dag Sjoberg,
„Evaluating Pair Programming with Respect to System Complexity and Programmer Expertise“,
IEEE Trans. On Software Engineering, Vol 33, no 2, Feb. 2007, 65-85.

Auswirkung der PP



Unterscheidung nach Kompetenz der Programmierer



Fazit

- Große Studie, mit fast 300 professionellen Teilnehmern.
- Berücksichtigte Systemkomplexität und Kompetenz.
- Ergebnisse:
  - PP ist effektiv für Anfänger, besonders wenn das zu ändernde System komplex ist.
  - PP ist ineffektiv für fortgeschrittene Programmierer (ohne Erfahrung mit PP).
  - Empfehlung: benutze Paare bei Wartung durch Anfänger.

- Sind Studenten als Teilnehmer brauchbar, wenn man auf Profis generalisieren möchte?

Einige Ergebnisse aus Experimenten

- Inspektion sind effektiv bei der Defekt-Eliminierung.
- Entwurfsmuster funktionieren wie beworben.
- Vererbungstiefe ist schlechter Prädiktor für Wartungsaufwand.
- Paarprogrammierung hilft nur Anfängern.
- Paarprogrammierung kann durch Einzelprogrammierung mit Reviews ersetzt werden (bei Anfängern).
- Test-getriebene Entwicklung bringt nichts.
- UML bringt für die Wartung nichts.

Beachte: alles Prozessexperimente, Erlernen geht rasch. Kaum Werkzeugentwicklung nötig.

Wann benutzt man das Experiment?

- Vorteile:
  - Kann Ursache-Wirkungs-Zusammenhang identifizieren
  - Experimentelle Methodik exzellent entwickelt (Statistik, Kontrolle)
- Nachteile:
  - Aufwändig, teuer
  - Professionelle Teilnehmer sind schwierig zu kriegen
  - Experimente dauern lange (1 Jahr pro Experiment)
  - Negative Ergebnisse sind häufig
  - Nur brauchbar, wenn Methodik/Werkzeug ausgereift und Erlernbarkeit kurzfristig möglich.
- Welche empirische Methode soll man nehmen, wenn eine Technik erst noch entwickelt und verbessert werden soll? Dafür ist ein Experiment meist zu teuer.

Ex post facto Studien: Analysieren von Software-Depots

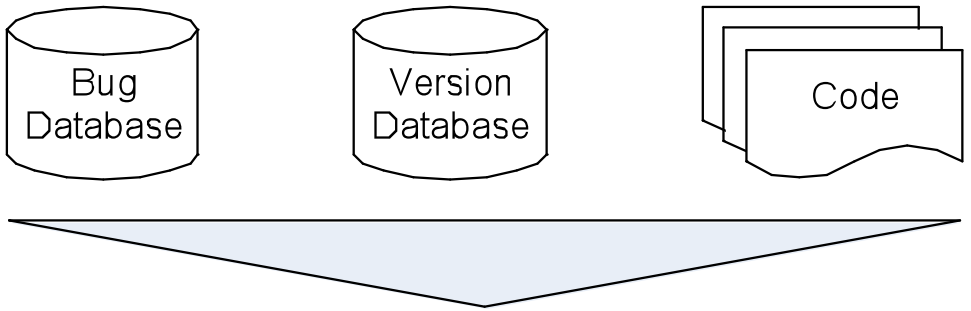
- Untersuche die Versionshistorie von Software in Verbindung mit Fehlermeldung
- Beispiel: Können Softwaremetriken fehleranfällige Komponenten vorhersagen?

Nagappan, Ball, Zeller: Mining Metrics to Predict Component Failures, ICSE 2006

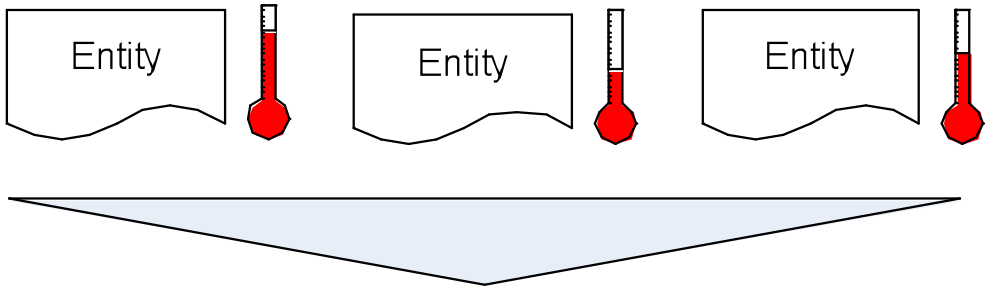
Zimmermann et al: Cross-project Defect Prediction, ESEC/FSE 2009.

High level description

1. Collect input data



2. Map post-release failures to defects in entities



3. Predict failure probability for new entities

Quelle: Nagappan

Projects researched

- Internet Explorer 6
- IIS Server
- Windows Process Messaging
- DirectX
- NetMeeting

>1,000,000 Lines of Code

Quelle: Nagappan

Per-function metrics — correlation with maximum and sum of metric across all functions *f*() in a module *M*							
Lines	# executable lines in *f*()	Max	-0.236	**0.514**	**0.585**	**0.496**	**0.509**
		Total	0.131	**0.709**	**0.797**	0.187	**0.506**
Parameters	# parameters in *f*()	Max	-0.344	0.372	**0.547**	0.015	0.346
		Total	0.116	**0.689**	**0.790**	0.152	0.478
Arcs	# arcs in *f*()'s control flow graph	Max	-0.209	0.376	**0.587**	0.527	0.444
		Total	0.127	**0.679**	**0.803**	0.158	**0.484**
Blocks	# basic blocks in *f*()'s control flow graph	Max	-0.245	0.347	**0.585**	0.546	0.462
		Total	0.128	**0.707**	**0.787**	0.158	0.472
ReadCoupling	# global variables read in *f*()	Max	-0.005	**0.582**	**0.633**	0.362	0.229
		Total	-0.172	**0.676**	**0.756**	0.277	0.445
WriteCoupling	# global variables written in *f*()	Max	0.043	**0.618**	**0.392**	0.011	0.450
		Total	-0.128	**0.629**	**0.629**	0.230	0.406
AddrTakenCoupling	# global variables whose address is taken in *f*()	Max	0.237	**0.491**	**0.412**	0.016	0.263
		Total	0.182	**0.593**	**0.667**	0.175	0.145
ProcCoupling	# functions that access a global variable written in *f*()	Max	-0.063	**0.614**	**0.496**	0.024	0.357
		Total	0.043	**0.562**	**0.579**	0.000	0.443
FanIn	# functions calling *f*()	Max	0.034	**0.578**	**0.846**	0.037	**0.530**
		Total	0.066	**0.676**	**0.814**	0.074	**0.537**
FanOut	# functions called by *f*()	Max	-0.197	0.360	**0.613**	0.345	0.465
		Total	0.056	**0.651**	**0.776**	0.046	**0.506**
Complexity	McCabe's cyclomatic complexity of *f*()	Max	-0.200	0.363	**0.594**	0.451	**0.543**
		Total	0.112	**0.680**	**0.801**	0.165	**0.529**

Metrics and their Correlation with Post-Release Defects

Per-class metrics — correlation with maximum and sum of metric across all classes *C* in a module *M*							
ClassMethods	# methods in *C* (private / public / protected)	Max	0.244	**0.589**	**0.534**	0.100	0.283
		Total	**0.520**	**0.630**	**0.581**	0.094	0.469
InheritanceDepth	# of superclasses of *C*	Max	**0.428**	**0.546**	0.303	0.131	0.323
		Total	**0.432**	**0.606**	**0.496**	0.111	0.425
ClassCoupling	# of classes coupled with *C* (e.g. as attribute / parameter / return types)	Max	**0.501**	**0.634**	**0.466**	-0.303	0.264
		Total	**0.547**	**0.598**	**0.592**	-0.158	0.383
SubClasses	# of direct subclasses of *C*	Max	0.196	**0.502**	**0.582**	-0.207	0.387
		Total	0.265	**0.560**	**0.566**	-0.170	0.387

Quelle: Nagappan

Do metrics correlate with failures?

Project	Metrics correlated w/ failure
A	*#Classes* and 5 derived
B	almost all
C	all except *MaxInheritanceDepth*
D	only *#Lines (software was refactored if metrics indicated a problem)*
E	*#Functions, #Arcs, Complexity*

Do metrics correlate with failures?

Project	Metrics correlated w/ failure
A	*#Classes* and 5 derived
B	alm[illegible]all
C	[illegible]xcept *M*[illegible]*nheritanceD*[illegible]
D	[illegible]ly *#Lines*
E	*Functions* [illegible]*mpl*

YES

But only within a project

Quelle: Nagappan

Is there a set of metrics that fits all projects?

Project	Metrics correlated w/ failure
A	*#Classes* and 5 derived
B	almost all
C	all except *MaxInheritanceDepth*
D	only *#Lines*
E	*#Functions, #Arcs, Complexity*

Is there a set of metrics that fits all projects?

Project	Metrics correlated w/ failure
A	*#Classes* and 5 derived
B	lmost a
C	cep xInh nceDep
D	nly
E	*#Functio tArcs, Co*

NO

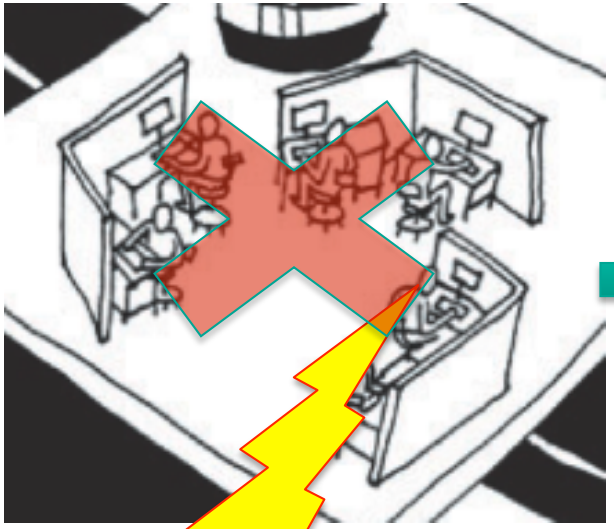
Quelle: Nagappan

Wann eignet sich die Analyse von Software?

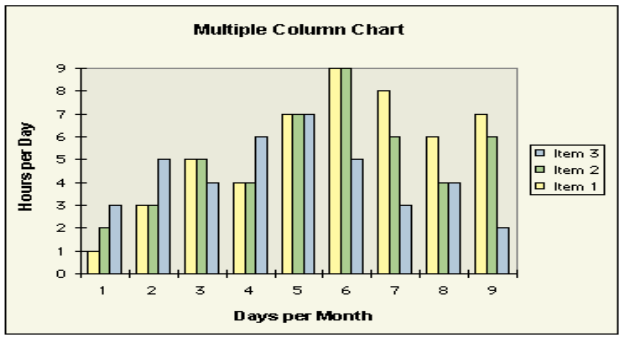
- Vorteile
  - Große Datenmengen verfügbar
  - Analyse automatisierbar
  - Kein Kontakt mit menschlichen Subjekten nötig ☺
- Nachteile
  - Man kann nur analysieren, was gegeben ist—wenn z.B. kein Paarprogrammieren durchgeführt wurde, kann man es auch nicht analysieren.
  - Ihre neueste Technik kennt noch keiner, daher gibt´s dafür auch keine Daten.
  - Man erhält Korrelationen, aber keinen sicheren Ursache-Wirkungs-Zusammenhang.

Analyse von Software-Depots

variiere unabhängige Variablen



kontrolliere Störvariablen



Beobachte abhängige Variablen

Was gibt´s denn noch?

Empirische Forschungsmethoden

Deskriptive Forschung
beschreibt Phänomene,
Ereignisse, Situationen

Experimentelle Forschung
Beschreibt Ursache-Wirkung,
ist quantitativ

Quantitative Studie

Umfrage
Korrelations-Studie
Ex post facto Studie
Langzeit- und Querschnittsstudie
Naturalistische Beobachtung
Meta-Studie

Qualitative Studie

Fallstudie
Ethnographie
Phänomenologie

Laborexperiment
Feldexperiment
Simulation
Benchmark

Was sind die Probleme der heutigen SW-Empirie?

- Fortschritt ist zu langsam, Kosten zu hoch.
- Ergebnisse sind nicht konstruktiv.
- Der typische Informatiker will etwas konstruieren, nicht nur analysieren.

- Wie kann man nachweisen, dass/ob neue Softwaretechniken besser sind,
- Softwaretechniken weiter verbessern (nachweislich),

- und das **schneller** als bisher?

Vorschlag: Benutze Benchmarks!

- Benchmarks sind Mengen von Problemen mit einer Qualitätsmetrik für Lösungen.
  - Unabhängige Forschungsgruppen wenden ihre automatisierten Werkzeuge an und vergleichen ihre Ergebnisse mit denen von anderen.
  - Verbesserungen können ohne große Verzögerung implementiert werden, gehen in richtige Richtung.
  - Benchmarks haben einen riesigen Vorteil gegenüber Versuchen mit menschl. Teilnehmern: Sie können beliebig oft wiederholt werden.
  - Benchmarks müssen weiterentwickelt werden, um Überanpassung zu vermeiden.

Benchmarks sind effektiv, Forschung vorwärts zu treiben.

- **Rechnerarchitektur:** Benchmarks werden seit Dekaden zum Leistungsvergleich benutzt.
  - Standard Performance Evaluation Corporation (SPEC) publiziert eine Reihe von Benchmarks (CPU, Web server, Mail Server, AppServer, power consumption, etc.)
  - Benchmarks und Simulation haben Rechnerarchitektur quantitativ gemacht.
  - Jede Prozessoreigenschaft muss sich an Benchmarks bewähren.
- Marketing Benchmarks: um die zu vermeiden, wurden unabhängige Benchmark-Organisationen gegründet.

Wo Benchmarks regieren:

- **Databanken:** Transaction Processing Performance Council (TPC)
- **Spracherkennung:** große Mengen von Sprachproben werden benutzt, um Spracherkenner zu trainieren und zu vergleichen (Fehlerraten) (DARPA)
- **Sprachübersetzung:** genauso.

Autonome Fahrzeuge: DARPA Herausforderungen

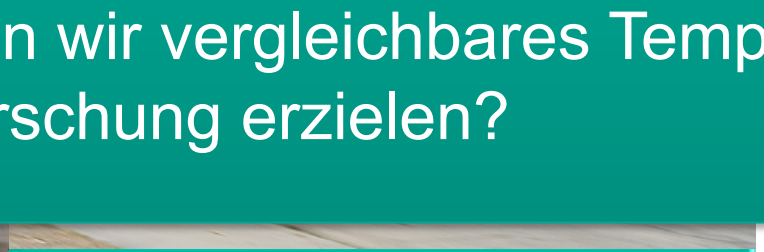
Google automes Fahrzeug

2007 DARPA
Urban Challenge

2004, 2005 DARPA
Grand Challenge

Autonome Fahrzeuge: DARPA Herausforderungen

2004, 2005 DARPA
Grand Challenge



2007 DARPA
Urban Challenge

In all diesen Fällen haben Benchmarks zu raschen und substanziellen Fortschritten geführt.

Erfolgreiche Techniken wurden von anderen Teams rasch aufgegriffen und verbessert.

Wie könnten wir vergleichbares Tempo in der Softwareforschung erzielen?

Software-Forschung könnte mehr Benchmarks benutzen

- Benchmarks können überall da angewandt werden, wo Softwareprozesse automatisiert werden.
- Beteilige dich an der Entwicklung brauchbarer Benchmarks, damit
  - die Arbeit auf mehrere Schultern verteilt wird,
  - bessere Werkzeuge gebaut werden können,
  - die besten Techniken herausgefiltert werden,
  - der Fortschritt beschleunigt wird.

- Beispiele ungewöhnlicher Benchmarks in der SWT folgen.

Beispiel: Verarbeitung natürlichsprachlicher Anforderungen

- Problem: übersetze sprachliche Anforderungen in UML Modelle (und zurück).
- Über 70% der Anforderungen sind in uneingeschränkter, natürlicher Sprache geschrieben.
- Benchmarks: Echte Anforderungen sind erstaunlich schwierig zu finden:
  - Lehrbücher enthalten wenige Beispiele, die die Autoren selbst erfunden haben, oder von anderen kopiert haben, die sie auch erfunden haben.
  - Firmen wollen keine freigeben (Qualität oft peinlich).
- Mit Beispielen könnten wir ein Problem nach dem anderen anpacken.
- Unsere Sammlung: http://nlre.wikkii.com/wiki/The_NLRP_Benchmark_Homepage

Ansatz: Thematische Rollen

Einige für SW-Ingenieure interessante Rollen:

- agens (AG) – der Handelnde.
  patiens (PAT) – der Behandelte.
  actus (ACT) – die Handlung (für Tätigkeitsverben und deren Nominalisierungen)

  Peter_{AG} wirft_{ACT} einen Ball_{PAT}.
  Peter_{AG} tätigt einen Wurf_{ACT} des Balls_{PAT}.

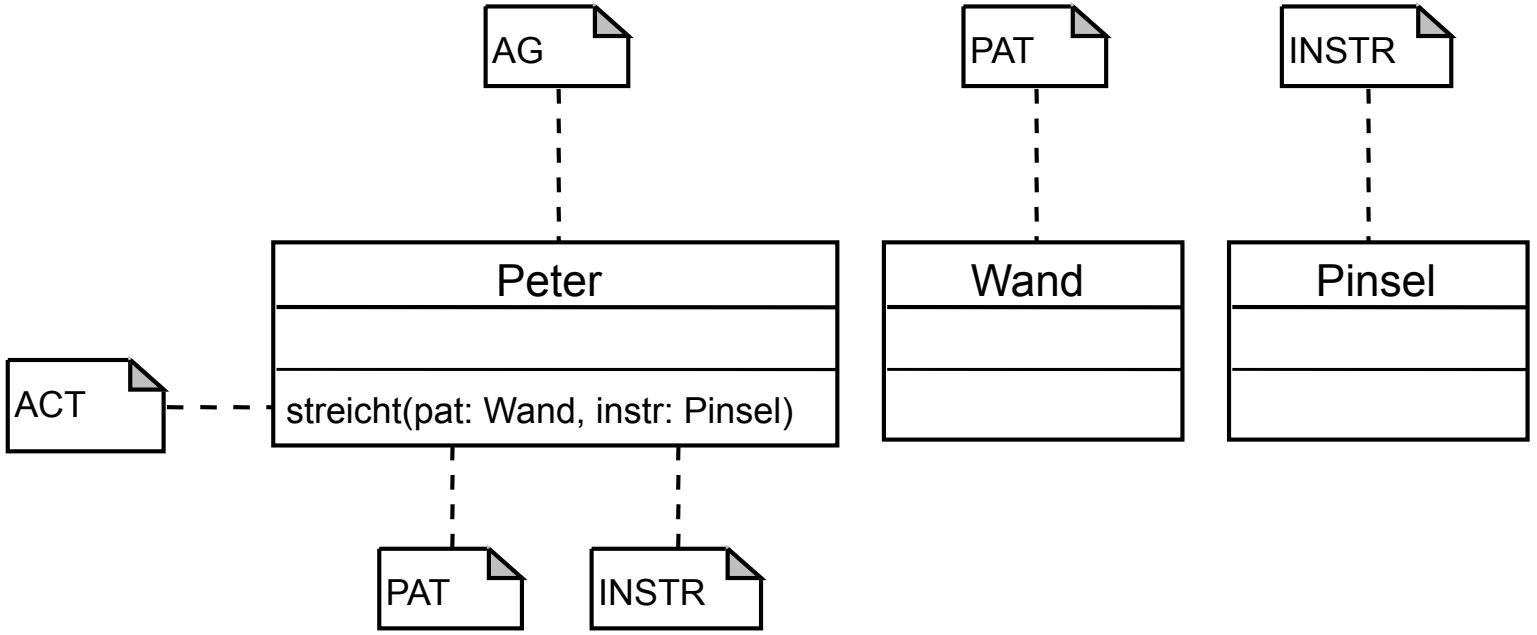
- instrumentum (INSTR) – das Hilfsmittel
  Peter streicht die Wand mit einem Pinsel_{INSTR}.

- status (STAT) – der Zustand (für Zustandsverben und deren Nominalisierungen)
  Peter wohnt_{STAT} in Bonn.

(Gelhausen2007)

Abbildung thematischer Rollen

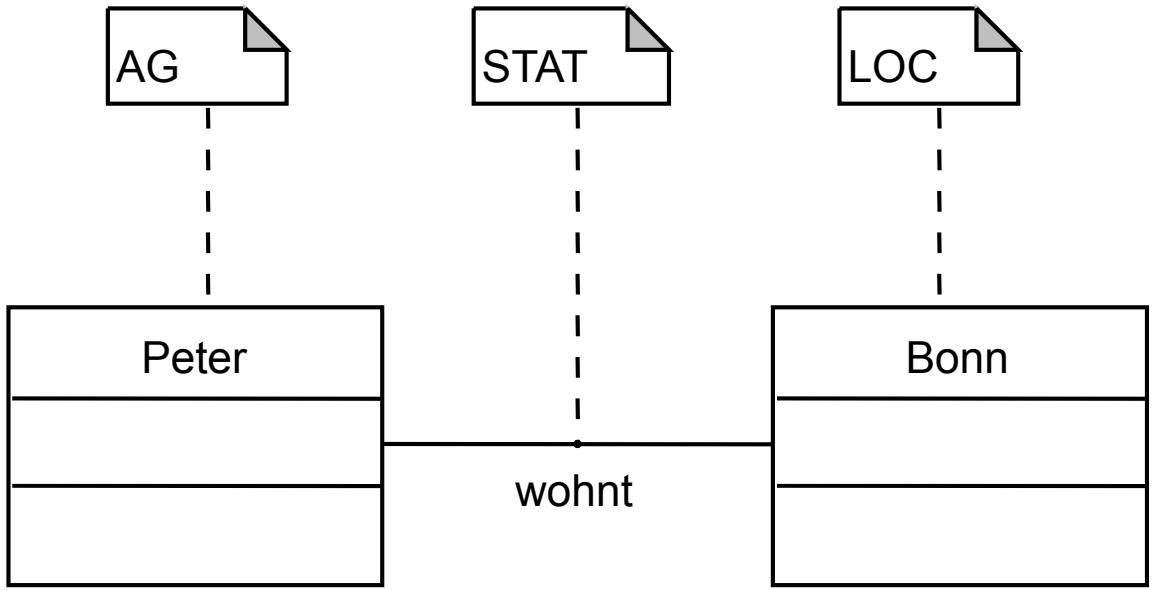
- „Peter_{AG} streicht_{ACT} die Wand_{PAT} mit einem Pinsel_{INSTR}.“
- Wir hätten auch schreiben können: „Die Wand_{PAT} wird von Peter_{AG} gestrichen_{ACT}, und zwar mittels eines Pinsels_{INSTR}.“



(Gelhausen2007)

Abbildung von Zustandsverben

- „Peter_{AG} wohnt_{STAT} in Bonn_{LOC}.“



Beispiel für eine einfache Relation.

(Gelhausen2007)

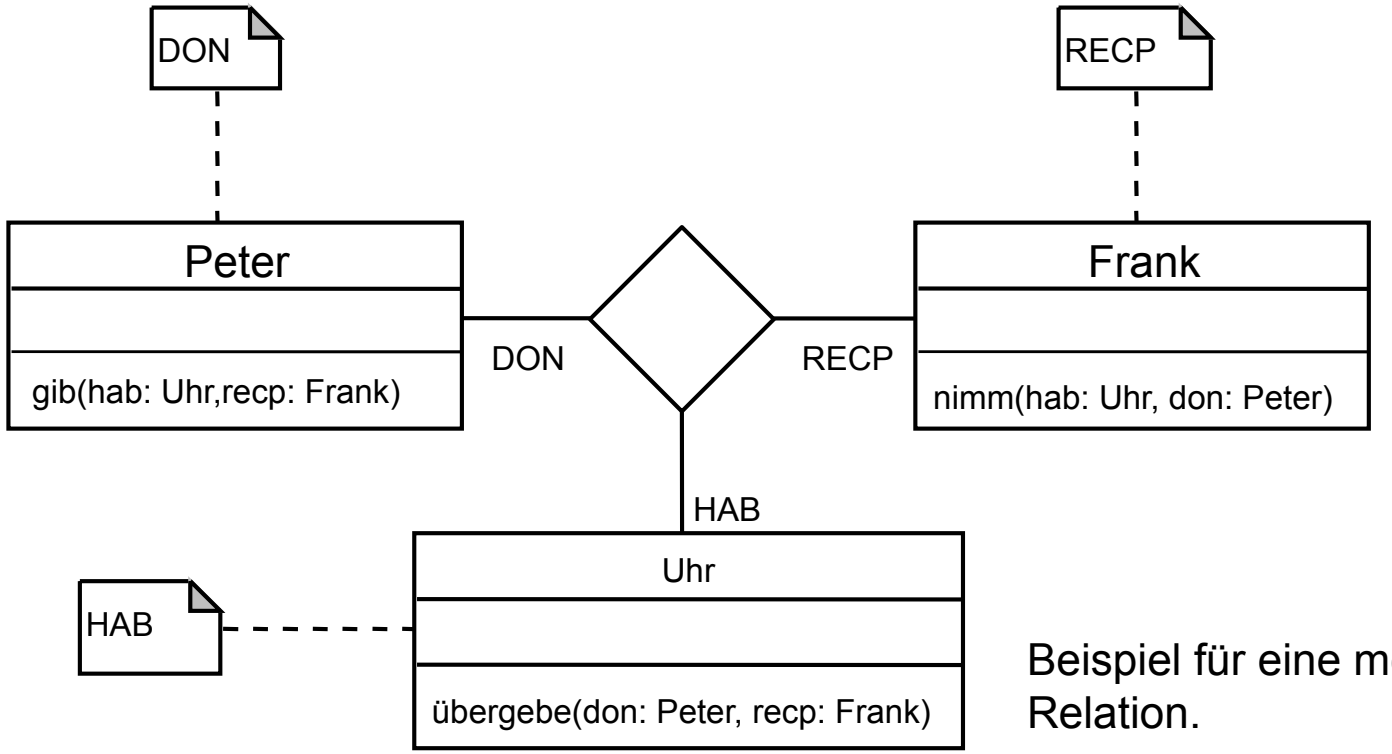
Thematische Rollen

- Einige für SW-Ingenieure interessante Rollen:
  - Donor (DON) – der, der etwas gibt
  - Recipient (RECP) – der Empfänger einer Sache.
  - Habitum (HAB) – etwas, das gegeben oder „gehabt“ wird.
    $\text{Peter}_{\text{DON}}$ schenkt $\text{Frank}_{\text{RECP}}$ eine Uhr_{HAB}.

(Gelhausen2007)

Abbildung auf Beziehungen

- „Peter$_{DON}$ schenkt Frank$_{RECP}$ eine Uhr$_{HAB}$.“



Beispiel für eine mehrstellige Relation.

(Gelhausen2007)

Etwas komplexeres Beispiel
Whois Protokoll

- Eine Spezifikation

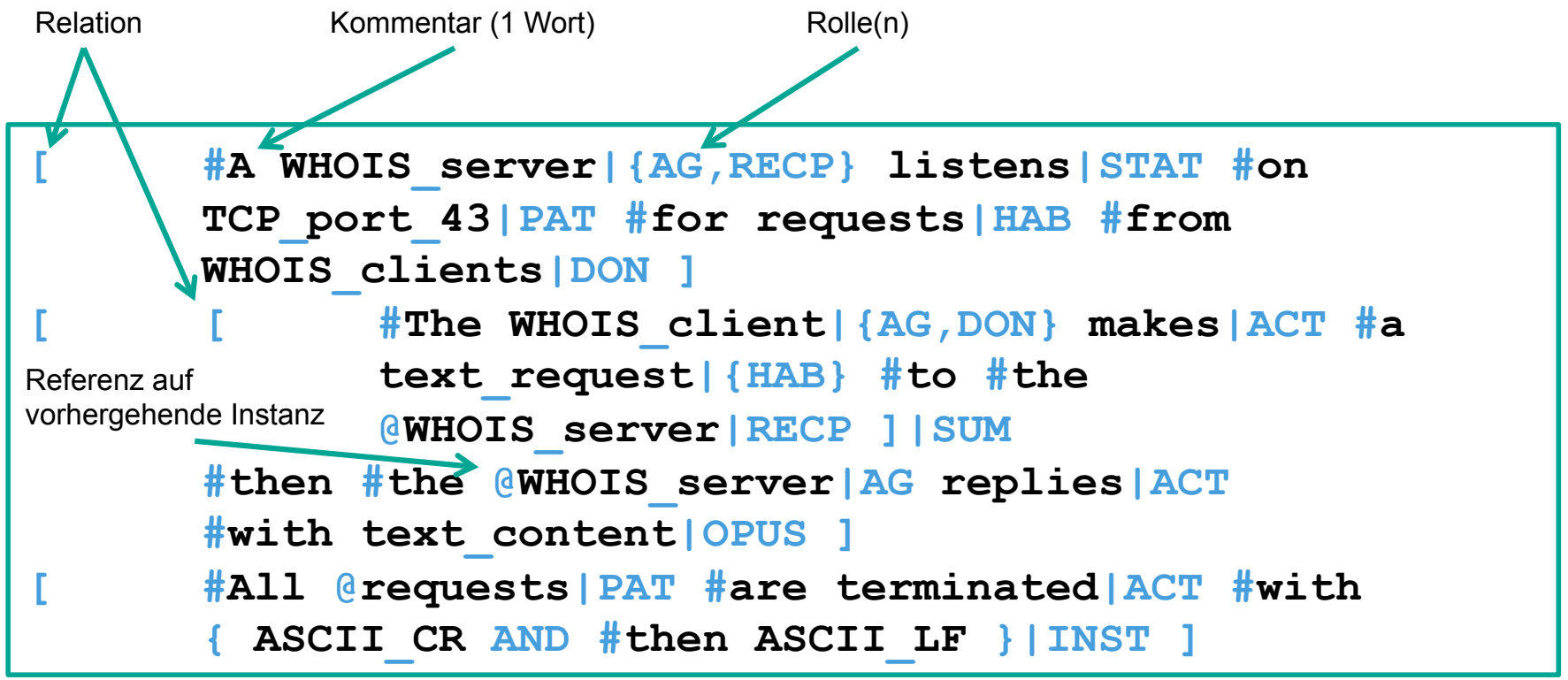
```
 A WHOIS server          listens       on
TCP port 43      for requests      from
WHOIS clients
        The WHOIS client          makes      a
       text request        to  the
        WHOIS server
 then  the  WHOIS server    replies
 with text content
 All  requests      are terminated      with
  ASCII CR      then ASCII LF
```

Quelle: IETF WHOIS Protocol Specification, RFC 3912, Ch. 2, “Protocol Specification”

(Gelhausen2007)

Etwas komplexeres Beispiel
Whois Protokoll annotiert

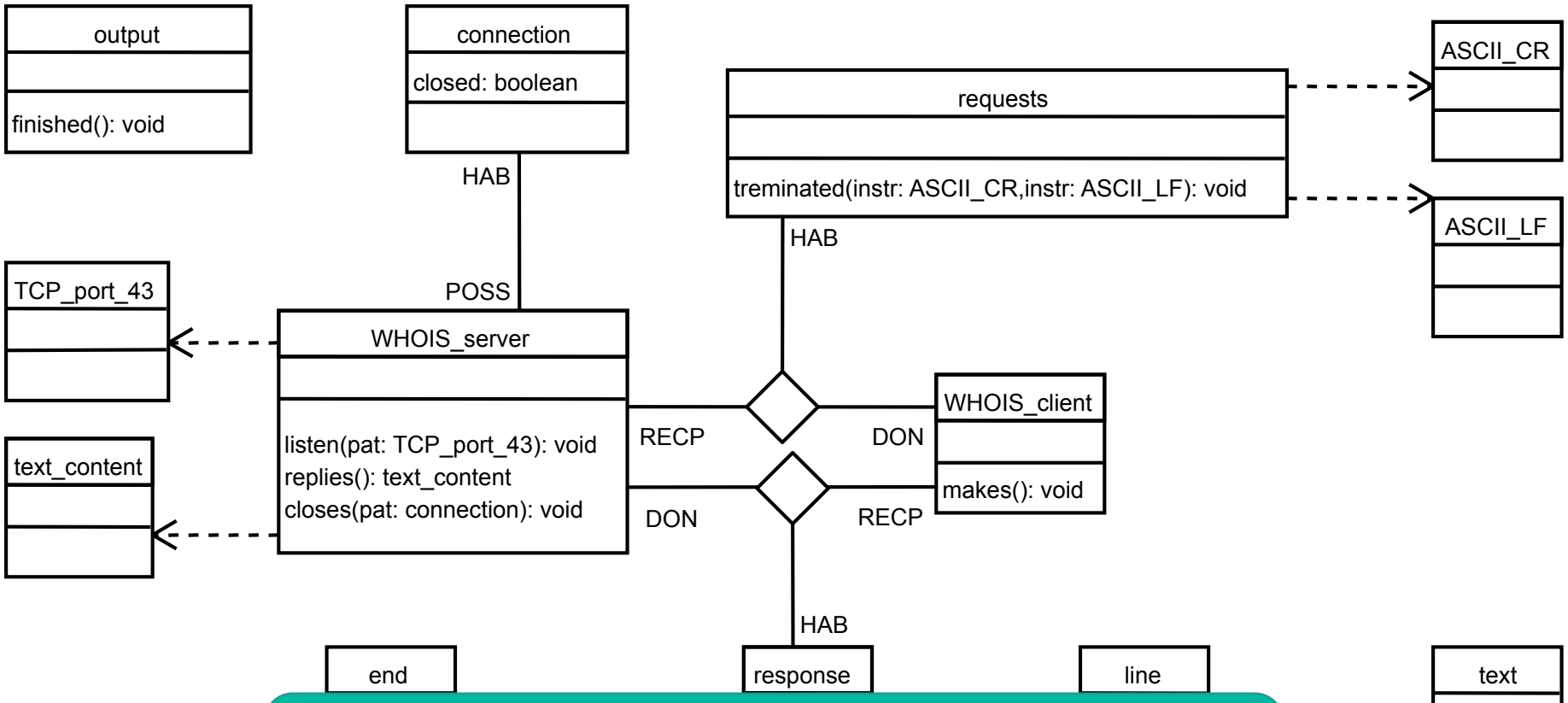
- Annotationssprache SAL_E



Quelle: IETF WHOIS Protocol Specification, RFC 3912, Ch. 2, “Protocol Specification”

(Gelhausen2007)

Whois Protokoll: Übersetzung in UML mit SAL_Emx



Qualitätsmetrik: Ausbeute, Präzision

(Gelhausen2007)

Beispiel: Auto-Parallelisierungs-Benchmark

- Um zu zeigen, dass automatische Parallelisierungstechniken funktionieren, stellen wir einen Benchmark zusammen.
- Besteht aus Paaren: seq. Implementierung und hand-parallelisierte Version (oder Versionen).
- Daran kann man z.B. das Konzept der Autofutures oder andere Parallelisierungsmuster testen.
  - Wurden alle Parallelisierungsmöglichkeiten gefunden?
  - Wurden sie richtig umgesetzt?
  - Wie schnell läuft die Parallelisierung?

Zusammenfassung

- Der Gebrauch von Benchmarks in der Softwaretechnik könnte deutlich höher sein (ist aber nicht die einzige Evaluierungstechnik).
- Mit realistischen Benchmarks bekäme man zuverlässige und vergleichbare Ergebnisse.
- Benchmarks beschleunigen den Fortschritt: sie sondern die schlechteren Ansätze aus, lenken die Aufmerksamkeit auf das Wichtige.
- Wenn die Werkzeuge ausgereift sind, dann erst überprüfe Brauchbarkeit mit Menschen (dem teuren Experiment).
- Literatur: Susan Elliott Sim, Steve Easterbrook, and Richard C. Holt. Using Benchmarking to Advance Research: A Challenge to Software Engineering, Proceedings of the Twenty-fifth International Conference on Software Engineering, Portland, Oregon, pp. 74-83, 3-10 May, 2003.

“If you are not keeping score, you’re just practicing.”

Vince Lombardi
Berühmter US Football Trainer

Barcelona gegen Manchester United: Wer spielt besser?