

Universität Karlsruhe (TH)

Forschungsuniversität · gegründet 1825

Kapitel 9.2

Agile Prozesse

SWT I – Sommersemester 2009

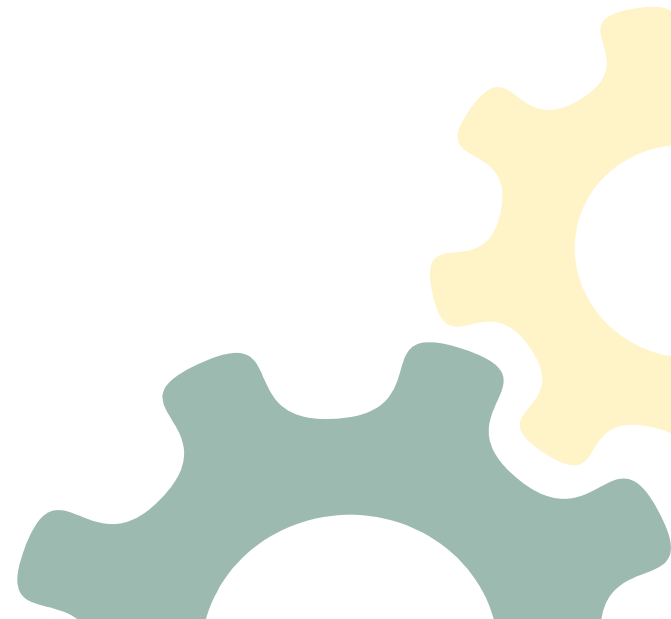
Andreas Höfer

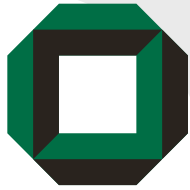
David J. Meder



Fakultät für Informatik

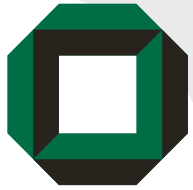
Lehrstuhl für Programmiersysteme





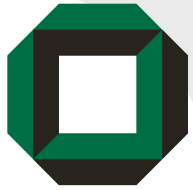
Agiles Manifest – eine Gegenreaktion zu plangetriebenen Prozessen

- Individuen und Interaktion wichtiger als Prozesse und Werkzeuge
- Laufende Software wichtiger als vollständige Dokumentation
- Kundenmitarbeit wichtiger als Vertragsverhandlungen
- Sich auf Änderungen einstellen wichtiger als Verfolgen eines Plans
- <http://www.agilemanifesto.org>



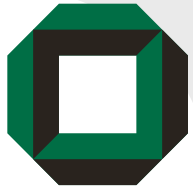
Agile Prozesse

- Minimum an Vorausplanung
- Planung erfolgt inkrementell
- Vermeiden unterstützender Dokumente
- Schnelle Reaktion auf Änderungen
- Einbeziehung des Kunden in die Entwicklung
- Vertreter: Extreme Programming (XP), Scrum, Crystal, Adaptive Software Development, Feature-Driven Development, Software-Expedition



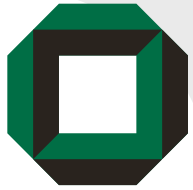
Besondere XP-Praktiken

- Paarprogrammierung
 - Testgetriebene Entwicklung
 - Inkrementeller Entwurf durch Umstrukturierungen
 - Iterative Planung in kurzen Zyklen (Planungsspiel)
 - Beteiligung eines echten Kunden
- Textuelle Beschreibung der Anwendungsfälle auf Karteikarten
 - Fortlaufende Integration
 - Gemeinsamer Code-Besitz
 - Programmierrichtlinien



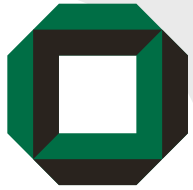
Paarprogrammierung I

- Jede Programmiertätigkeit wird im Paar ausgeführt
- Arbeit an einer Tastatur, einer Maus und einem Bildschirm
- Rollenverteilung: Fahrer und Beifahrer
 - Fahrer denkt an Implementierung des Algorithmus
 - Beifahrer denkt strategisch, führt ständige Durchsicht durch



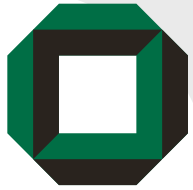
Paarprogrammierung II

- Führt zu nachweisbarer höherer Qualität des Programmtextes
- Strittig: Wirklicher Vorteil, Ressourcenauslastung



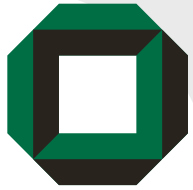
Effizientes Testen

- Möglichst zeitnah zur Programmierung
- Automatisiert und damit wiederholbar
- Soll Spaß machen
- Testen so oft und so einfach wie Übersetzen
- Fehler finden, nicht Fehlerfreiheit beweisen



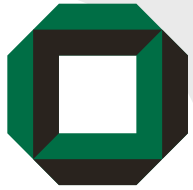
Testen bei XP

- *“Any program feature without an automated test simply doesn’t exist.” [Beck 2000]*
- Programmierer schreiben automatische Komponententests
- Kunde spezifiziert Akzeptanztests, die das Team implementiert
- Andere Testarten auch möglich, z. B. Stresstest
- Testausführung häufig und automatisch



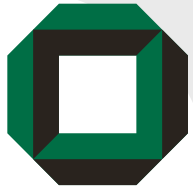
Komponententests

- Werden vom Entwickler selbst geschrieben
- Geben konkretes Feedback und Sicherheit
- Ermöglichen sichere Änderungen
- Sichern Erhalt der vorhandenen Funktionalität
- Müssen bei jeder Code-Integration zu 100 % laufen



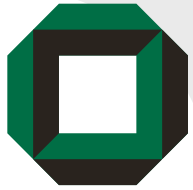
Akzeptanztests

- Werden vom Kunden spezifiziert
- In seltenen Fällen (mit Unterstützung durch Entwickler) vom Kunden selbst geschrieben
- Testrahmenwerke wie z. B. FIT(Nesse) oder das Robot Framework helfen dabei
- Müssen spätestens bei Auslieferung laufen
- Erhöhen (zusammen mit Komponententests) das Vertrauen des Kunden in das Produkt

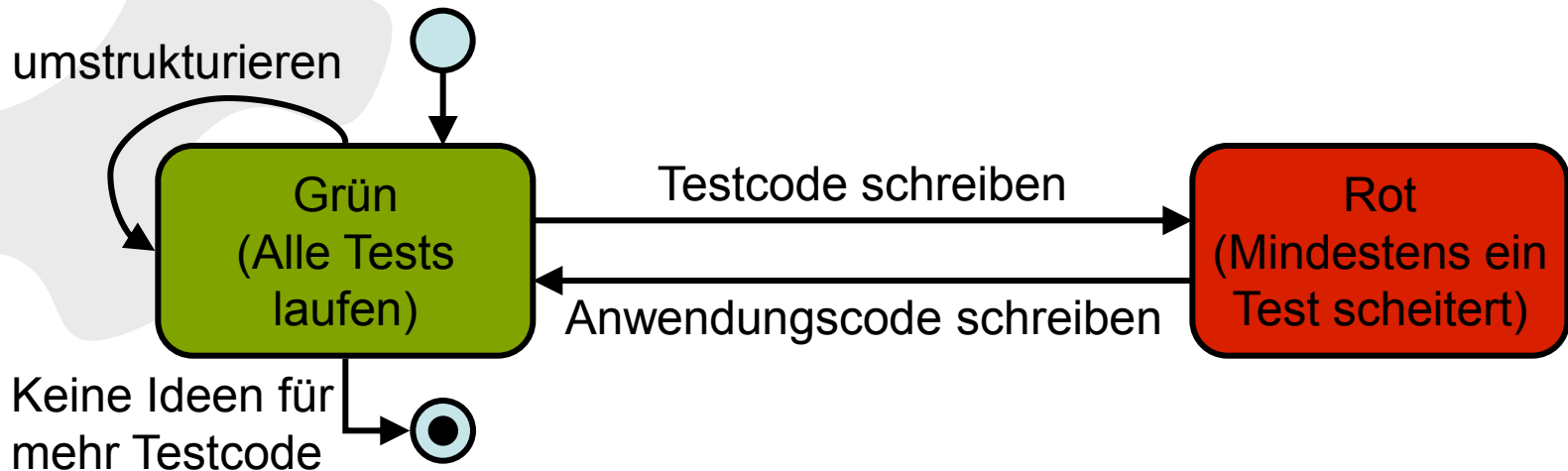


Testgetriebene Entwicklung

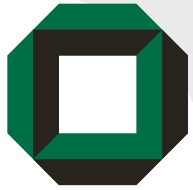
- Test/Implementierung/Umstrukturierung:
Motiviere jede Verhaltensänderung am Quelltext durch einen automatisierten Test.
- Umstrukturierung und einfacher Entwurf:
Bringe den Code immer in eine einfache Form.
- Häufige Integration:
Integriere den Code so häufig wie möglich.
- Paarprogrammierung:
Soll helfen die Regeln der testgetriebenen Entwicklung einzuhalten.



Zustandsdiagramm der testgetriebenen Entwicklung

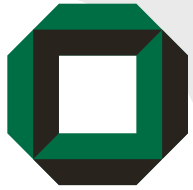


- Zustandsdiagramm idealisiert
- Testcode vor Anwendungscode
- kleine Schritte
- Inkrementeller Entwurf



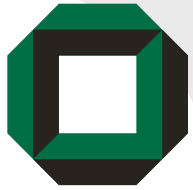
Test/Implementierung/ Umstrukturierung

- **grün** → **rot**:
Schreibe einen Test der fehlschlägt. Schreibe gegebenenfalls gerade soviel Quelltext, dass der Test übersetzt werden kann.
- **rot** → **grün**:
Schreibe gerade soviel Quelltext, dass alle Tests erfolgreich laufen.
- **grün** → **grün**:
Eliminiere Duplikationen und andere schlechte Eigenschaften im Quelltext.



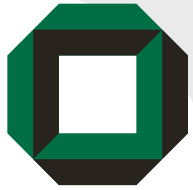
Rolle der Komponententests

- Tests zur Qualitätssicherung:
Die Tests sichern die vorhandene Funktionalität
- Test-Zuerst-Entwicklung führt zu „Evolutionärem Entwurf“: Der Entwurf entsteht Stück für Stück.
- Tests zur Schnittstellendefinition:
Test verwendet schon Klassen und Methoden, bevor sie überhaupt definiert sind.
- Tests zur Modularisierung:
Testbarkeit erfordert Entkopplung der Programmteile.
- Tests als ausführbare Spezifikation und Dokumentation



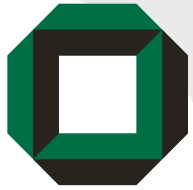
Umstrukturierung I

- Engl. Refactoring:
“A change made to the internal structure of software to make it easier to understand and cheaper to modify without changing its observable behavior.” [Fowler 1999]
- Voraussetzungen:
 - automatische Tests: sie stellen lauffähiges Produkt nach Umstrukturierung sicher
 - kollektiver Code-Besitz: Änderungen am ganzen Produkt für jeden möglich



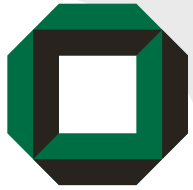
Umstrukturierung II

- Ziel: Stets eine möglichst einfache Form, die für die bisher implementierten Anforderungen ausreicht (kein änderungsfreundlicher, vorausschauender Entwurf nötig)
- Die einfache Form ist erreicht, wenn der Code...
 - alle seine Tests erfüllt.
 - vom „Zielpublikum“ verstanden werden kann.
 - jede Intention der Programmierer ausdrückt.
 - keine duplizierte Logik enthält.
 - möglichst wenig Klassen und Methoden umfasst.
 - Reihenfolge entscheidend!



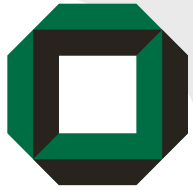
Umstrukturierung III

- Soll Produktivität erhöhen
- Durchgeführt an faulen Gerüchen (engl. bad smells) des Programmtextes, z. B.:
 - duplizierter Programmtext
 - parallele Vererbungsbäume
 - Schrotflintenchirurgie: eine neue Funktion erfordert Änderung an vielen Stellen
 - Kommentare zu umfangreich (Variablen besser benennen, Codestücke herausziehen und in Methoden verpacken)



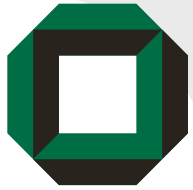
Inkrementelles Design

- Konventioneller Ansatz:
„Implement for today, design for tomorrow“
[unbekannt]
- XP Ansatz:
„The system should be designed as simply as possible at any given moment. Extra complexity is removed as soon as it is discovered.“
[Beck 2005]
- Kleine inkrementelle Designschritte statt großer Entwurfsphase



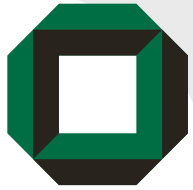
Planungsspiel

- Ziel: Planung von Umfang, Zeit und Kosten
 - der nächsten Iteration (wöchentlicher Zyklus)
 - der nächsten Auslieferung (vierteljährlicher Zyklus)
- Ständige Korrektur des Planes

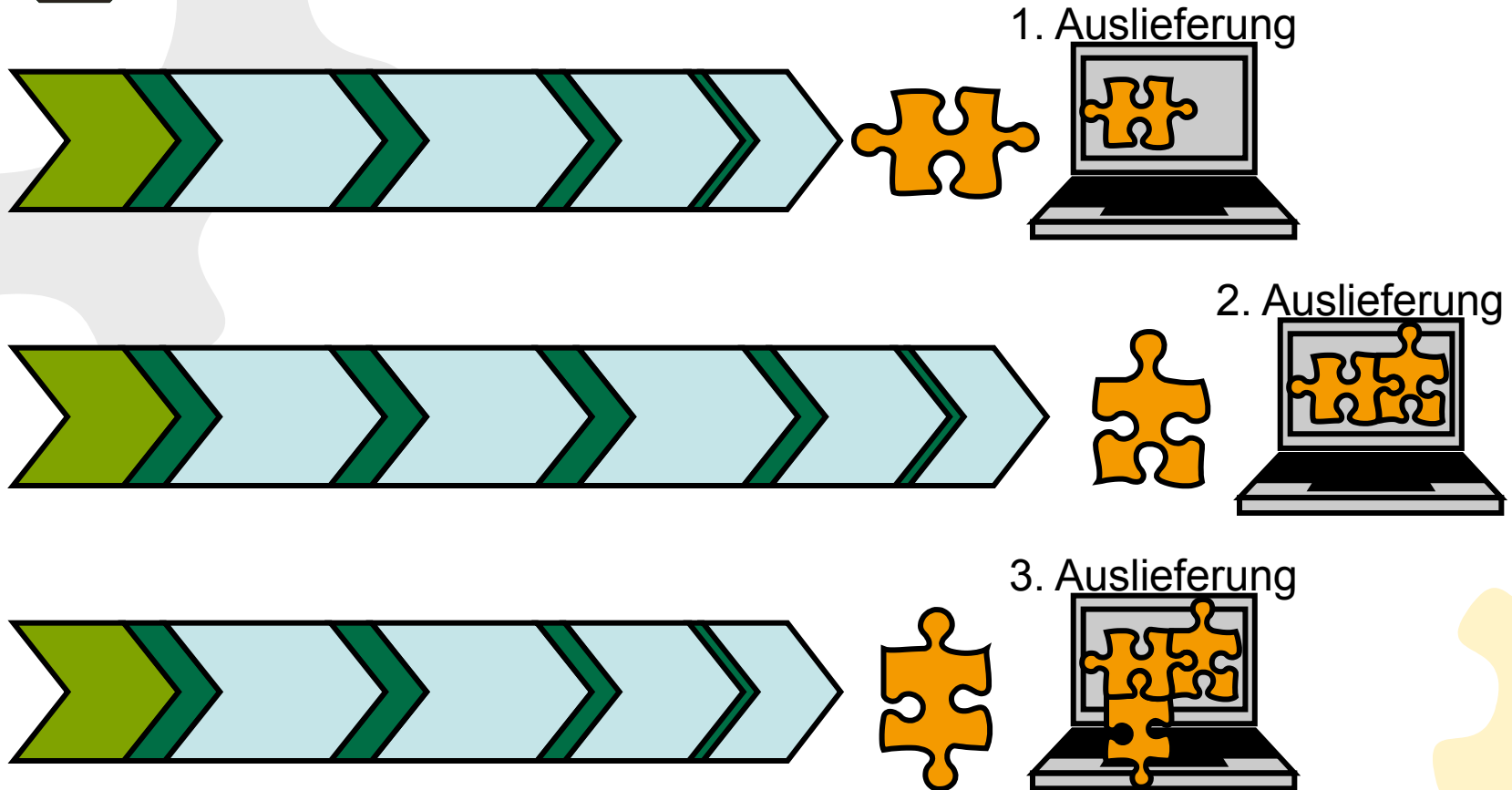


Kompetenzen beim Planungsspiel

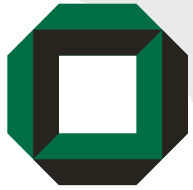
- Der Kunde trifft die geschäftsrelevanten Entscheidungen
 - Umfang des Systems
 - Prioritäten von Funktionalität
 - Zusammensetzung einer Auslieferung
 - Datum der Auslieferung
- Das Entwicklungsteam trifft die technischen Entscheidungen
 - Schätzung von Kosten und Risiko
 - Technische Konsequenzen
 - (Entwicklungs-) Prozess
 - Ablaufplanung
- Der Kunde kann seine Entscheidung jederzeit korrigieren



Planungsspiel Übersicht

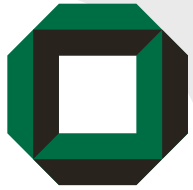


➤ Auslieferungsplanung ➤ Iterationsplanung ➤ Iteration



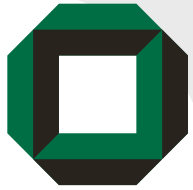
Auslieferungsplanung

- Auslieferungsplanung ersetzt konventionelle Anforderungsanalyse
- Planungszeitraum: 1 – 4 Monate
- Ungefährer Projektplan wird vom Kunden und vom Entwicklungsteam gemeinsam erstellt
 - Kunde äußert seine Wünsche (Anwendungsfall-Karten)
 - Team schätzt die Kosten und Risiken
 - Kunde priorisiert seine Wünsche
 - Gesamtes Team stellt Zeitplan für die wichtigsten Kundenwünsche auf



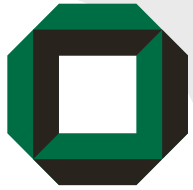
Iterationsplanung I

- Findet mehrmals bis zu einer Auslieferung
- Planungshorizont: 1 – 4 Wochen
- Alle Aufgaben werden auf Aufgaben-Karten notiert:
 - Aufteilung von Anwendungsfällen in kleinere Aufgaben
 - Zusammenlegen zu kleiner Aufgaben
 - Identifizierung zusätzlicher Aufgaben
 - Prototypen
 - Versionshaltung
 - etc.



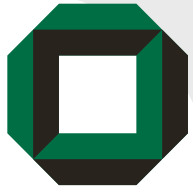
Iterationsplanung II

- Teammitglied akzeptiert Aufgabe und schätzt Aufwand für diese Aufgabe
- Ausgleich zwischen Teammitgliedern
- Rückkopplung zum Kunden



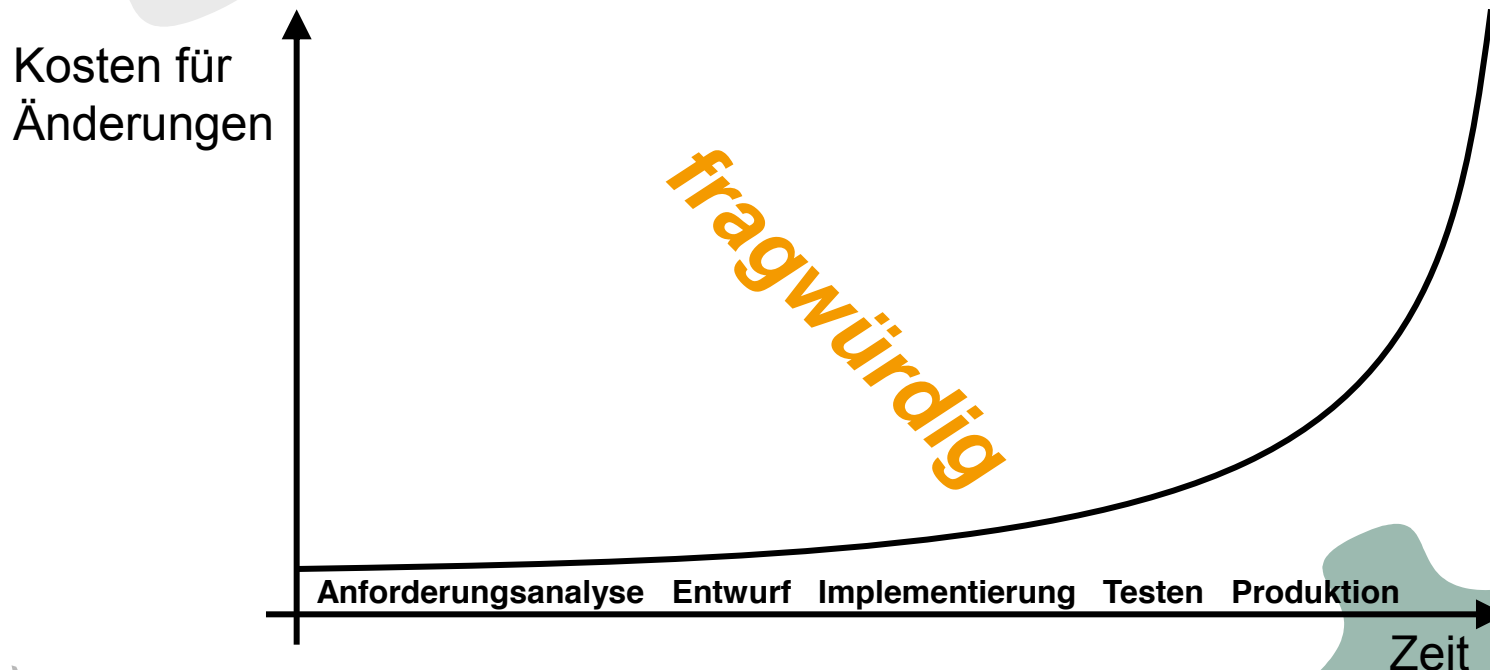
Kunde vor Ort

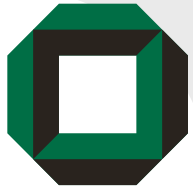
- Echter Kunde ist jemand, der das Produkt später wirklich nutzen soll
- Kunde sollte ständig verfügbar sein
- Der Kunde sollte im selben Raum arbeiten wie Entwicklerteam



Behauptung Kent Becks I

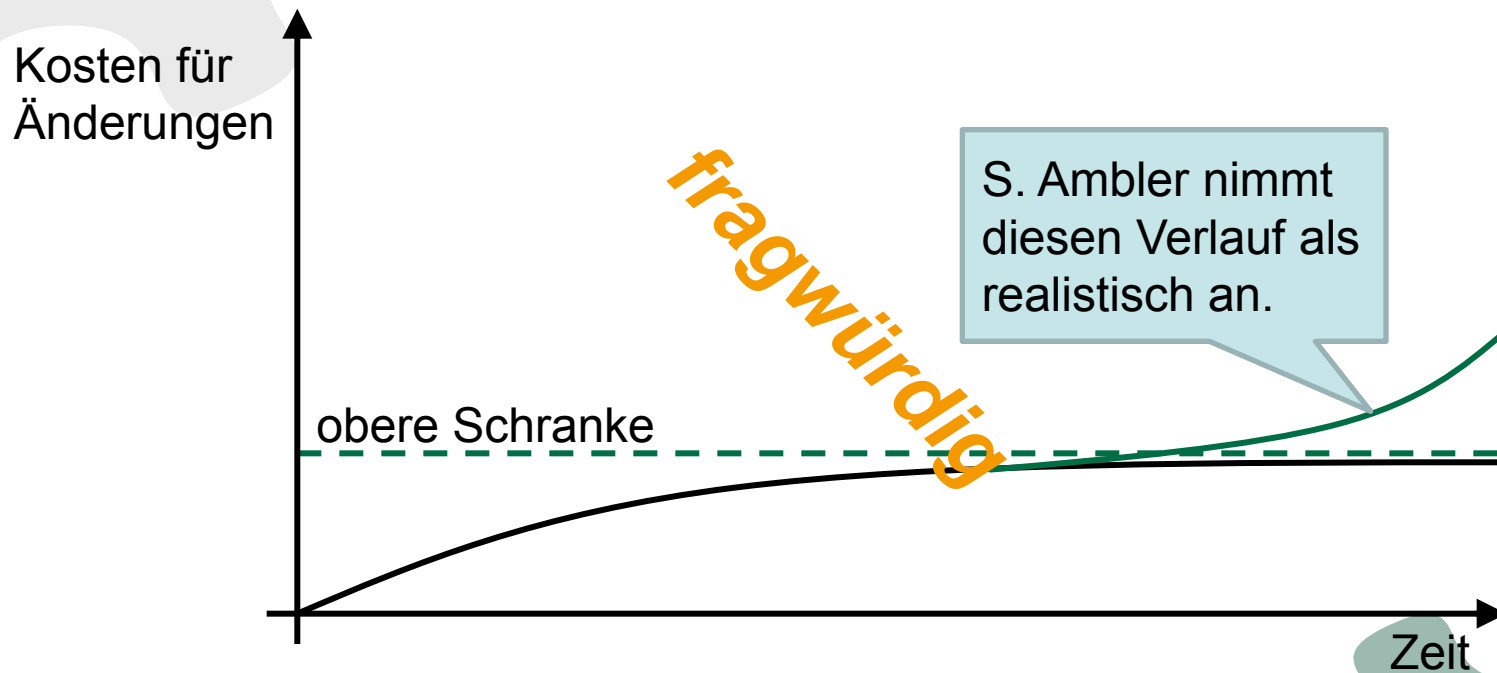
Bei konventionellem Entwicklungsprozess steigen die Kosten für Änderungen exponentiell mit der Zeit.

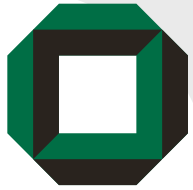




Behauptung Kent Becks II

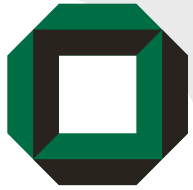
Für den XP Entwicklungsprozess verläuft die Kurve dagegen asymptotisch.





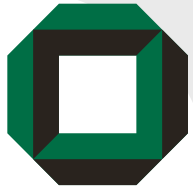
Kritik an XP

- Asymptotische Kostenkurve beruht auf der Erfahrung einzelner
- Fehlende Produktdokumentation
- Nicht reproduzierbarer ad-hoc Prozess
- Paarprogrammierung kostenintensiv, unklar ob dies durch den Zuwachs an Qualität kompensiert wird



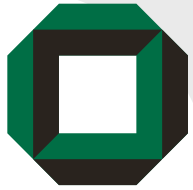
Probleme bei der Einführung von XP

- Kunde muss mitarbeiten und dies auch wollen
- Softwareingenieure sind auf Vorausschau getrimmt. Folge kann sein:
 - Zu komplexer Entwurf
 - Zu umfangreicher Entwurf
- Testgetriebene Entwicklung erfordert umdenken und umlernen
- Gleiche Wellenlänge bei Paarprogrammierung
- Respekt vor fremdem Programmtext



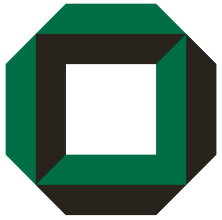
Bewertung von XP

- XP = Versuch und Irrtum?
- Nein,
 - Planungsschritte
 - Forderung nach erfolgreichen Tests
- Cowboy-Programmierer diszipliniert
- Spektrum des Planens
 - XP leichte Planung
 - Wasserfallmodell



Zusammenfassung

- XP ist Softwareprozess
 - Geeignet für vage und sich schnell ändernde Anforderungen
 - Für kleines Entwicklerteam
 - Ohne zeitraubenden Verwaltungsaufwand
- Softwareentwicklung mit leichtem Gepäck



Universität Karlsruhe (TH)

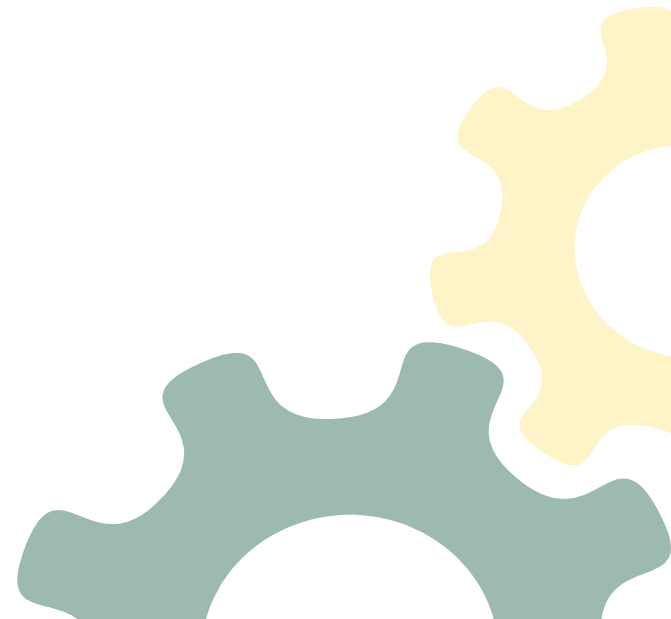
Forschungsuniversität · gegründet 1825

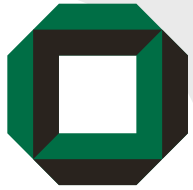
Was wissen wir eigentlich über XP?



Fakultät für **Informatik**

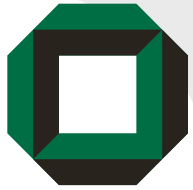
Lehrstuhl für Programmiersysteme





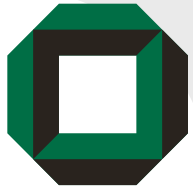
Paarprogrammierung versus...

- Partnerprogrammierung
- Einzelprogrammierung
- Einzelprogrammierung mit Durchsichten
 - Müller 2004: Are Reviews an Alternative to Pair Programming? Erschienen in Empirical Software Engineering Band 9, Ausgabe 4 (Dezember 2004) S. 335 - 351



Motivation

- Gibt es eine Methode, die (fast) das Gleiche leistet wie Paarprogrammierung aber mit geringeren Kosten?
- Wie verhalten sich Durchsichten gegenüber Paarprogrammierung bezüglich Kosten und Korrektheit?



Hypothesen

Merke:

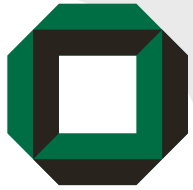
Korrektheit und Kosten
werden verglichen.

- Null-Hypothesen

- $H0_{\text{Korrektheit}}$: Die Korrektheit der mit Einzelprogrammierung und Durchsicht (ED) erstellten Programme ist höher oder gleich derer, die mit Paarprogrammierung (PP) erstellt wurden.
- $H0_{\text{Kosten}}$: Die Kosten für PP sind geringer oder gleich als die Kosten für ED.

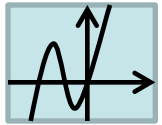
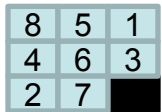
- Forschungs-/Alternativhypothesen

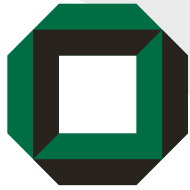
- $H1_{\text{Korrektheit}}$: Die Korrektheit der mit ED erstellten Programme ist geringer derer, die mit PP erstellt wurden.
- $H1_{\text{Kosten}}$: Die Kosten für PP sind höher als die Kosten für ED.



Übersicht

- Experiment durchgeführt SS 2002/03
- Gegenbalancierter Experimententwurf
- Teilnehmer: Studenten des XP Praktikums
- Programmiersprache: Java
- Vorstellung von Paarprogrammierung und Durchsichten
- Aufgaben:
 - Lösen eines Schiebepuzzles (SP)
 - Berechnen der Nullstellen eines Polynoms 3. Grades (Pol)

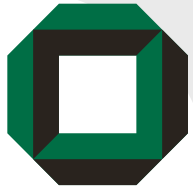




Gegenbalancierter Entwurf

Gruppe	1. Aufgabe	2. Aufgabe	Datenpunkte	
	Methode, Problems.	Methode, Problems.	PP	ED
1	PP, SP	ED, Pol	3	6
2	PP, Pol	ED, SP	2	3
3	ED, SP	PP, Pol	3	5
4	ED, Pol	PP, SP	2	3
		Summe:	10	17

- Erhalte zwei Datenpunkte pro Teilnehmer
- Kombiniere jede Methode mit jeder Problemstellung
- Stelle jede Kombination aus Methode/
Problemstellung an jede Stelle im Versuchsplan



Experimentplan

Austeilen der Problemstellung

„Ich bin fertig!“

Programm besteht 95 % der Testfälle.

Entwicklungsphase

Implementierung

Durchsicht

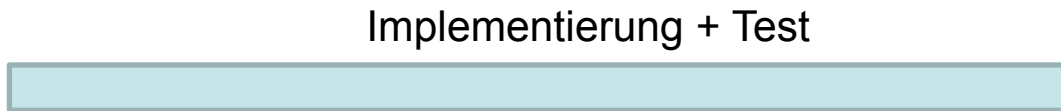
Test

Qualitätssicherungsphase

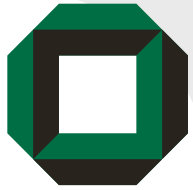
Einzelprog.
+ Durchsicht
(ED)



Paarprog.
(PP)

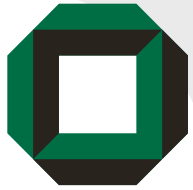


1. Akzeptanztest letzter Akzeptanztest

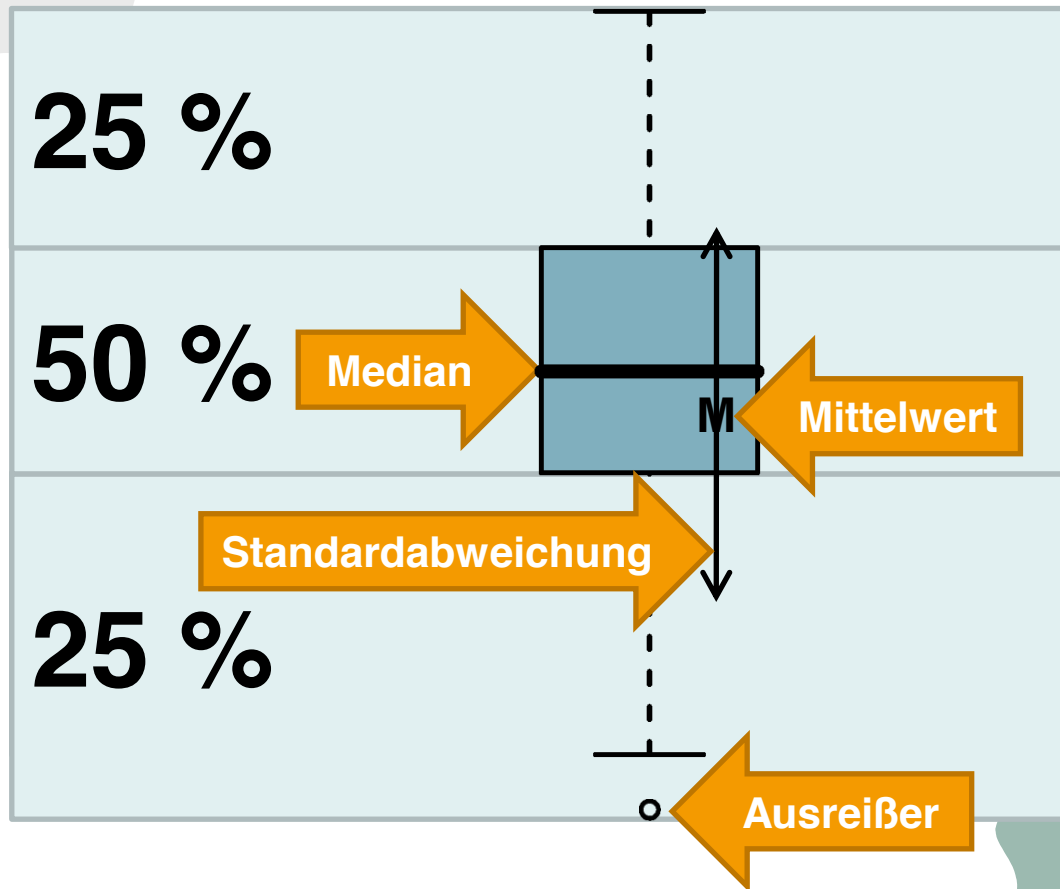


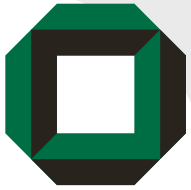
Warum eine Qualitätssicherungsphase?

- Programme vor Qualitätssicherungsphase durch Faktor „Sorgfalt beim Testen“ beeinflusst.
- Schlechte Programme kann man schnell schreiben.
- Die Qualitätssicherungsphase gleicht die Korrektheit der Programme an.
- Damit ist der Vergleich von etwa „gleich guten“ Programmen möglich.
- Hilfsmittel: automatisierter Akzeptanztest

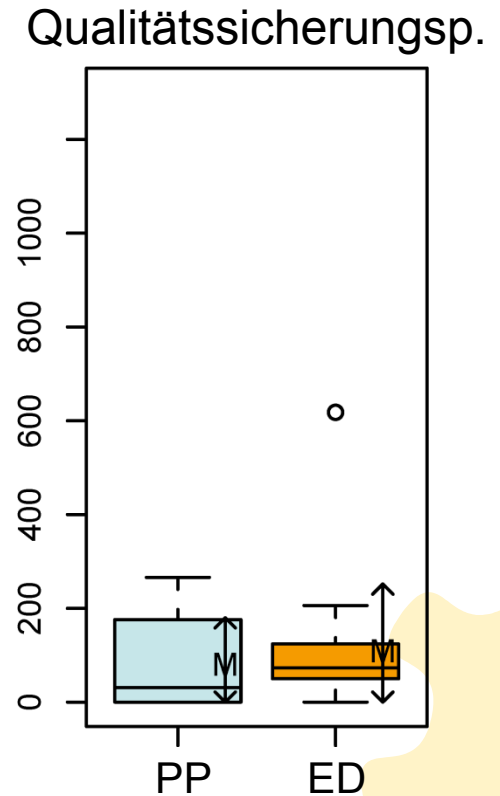
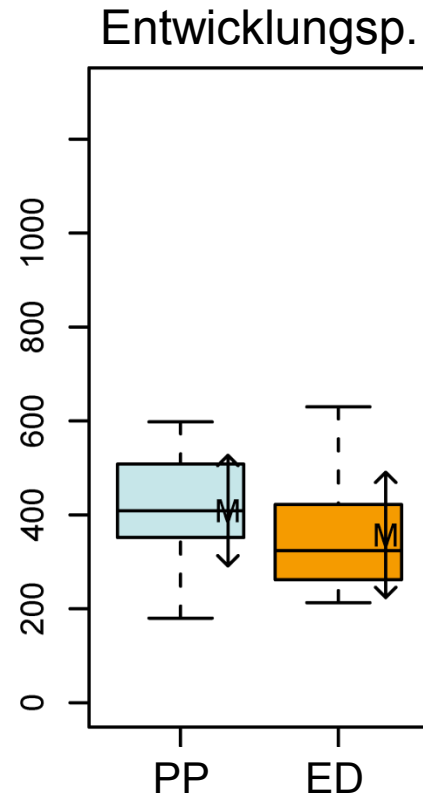
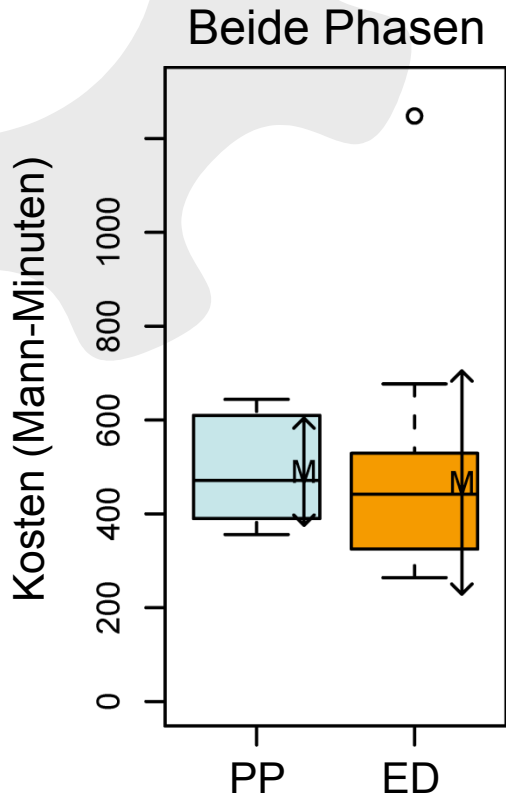


Einschub: Boxplots

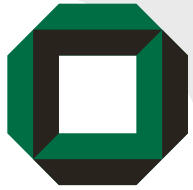




Ergebnisse Kosten

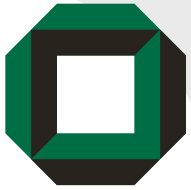


Kosten = Zeit für Phase(n) × Anzahl Personen



Ergebnisse Kosten II

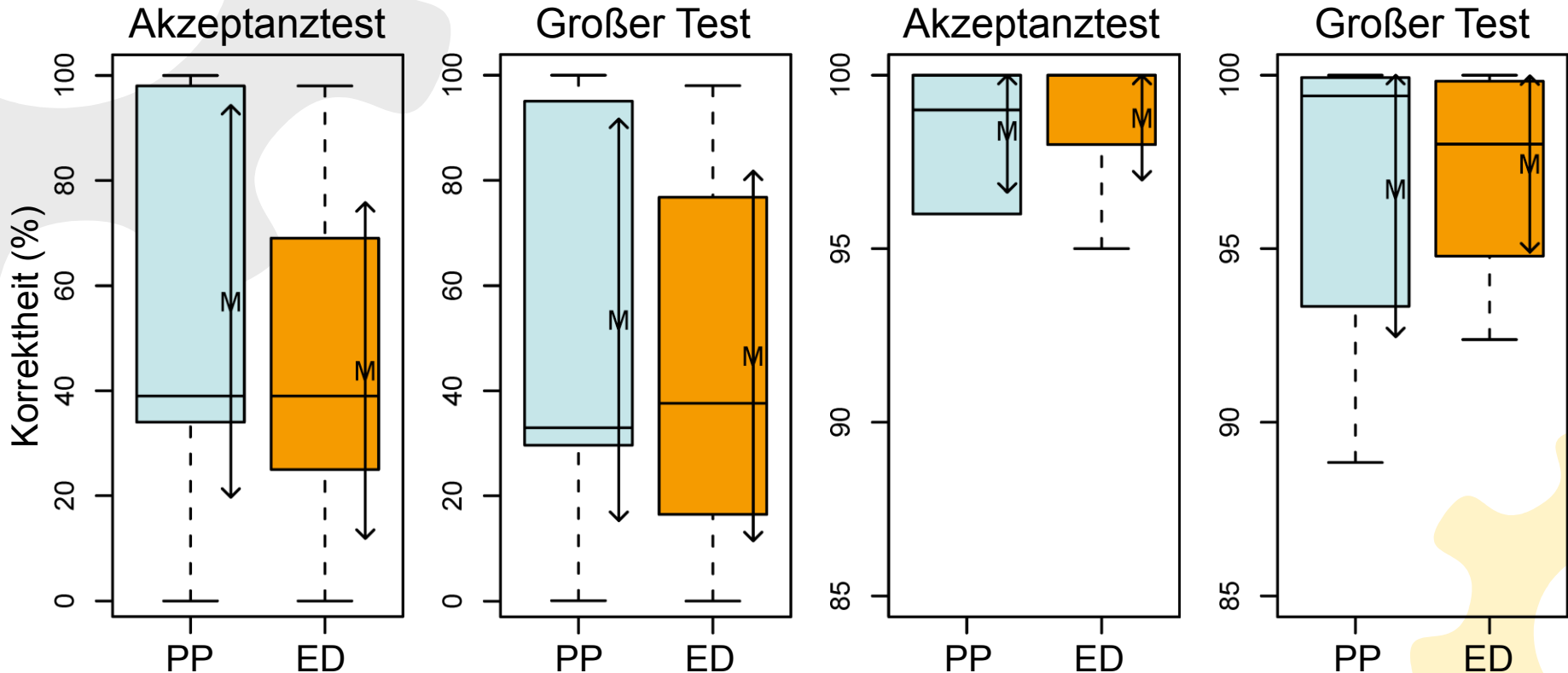
- Gesamt: Paare sind etwa 7 Prozent teurer bei vergleichbarer Korrektheit
- Implementierung: Paare sind etwa 13 Prozent teurer, entwickeln aber Programme, die 29 Prozent mehr Testfälle bestehen
- Unterschiede statistisch nicht signifikant



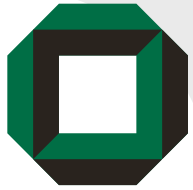
Ergebnisse Korrektheit

Vor erstem AT

Nach letztem AT

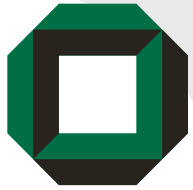


$$\text{Korrektheit} = \# \text{ bestandene Testfälle} / \# \text{ Testfälle} \times 100$$



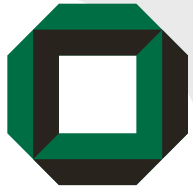
Ergebnisse Korrektheit II

- Qualitätssicherungsphase erfüllt ihren Zweck: vergleichbare Korrektheit der Programme
- Programme der PP Gruppe nach Entwicklungsphase (ohne Qualitätssicherungsphase) tendenziell besser; 29% mehr Testfälle bestanden.
- Unterschiede statistisch nicht signifikant



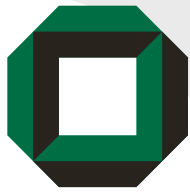
Fazit

- Keine der Null-Hypothesen kann abgelehnt werden.
- Einzelprogrammierung mit Durchsichten scheinen tendenziell etwas billiger zu sein.
- Paarprogrammierung und Einzelprogrammierung mit Durchsichten sind austauschbar.



Weitere quantitative Ergebnisse zur Paarprogrammierung

- Arisholm et al. 2007
 - Teilnehmer: 295 Java Berater
 - Aufwand für Paarprogrammierung ist 84 % höher als für Einzelprogrammierung
 - Unterschied zwischen Einzel- und Paarprogrammierung bzgl. Dauer, Aufwand und Korrektheit der Programme hängt von der Systemkomplexität ab
 - Beim komplexem System lieferten die Paare 48 % mehr korrekte Lösungen.
 - Einfluss der Kompetenz der Programmierer auf die Effizienz der Paarprogrammierung konnte nicht nachgewiesen werden
- Dybå et al. 2007
 - Meta-Studie („Studie über Studien“)
 - Fasst 15 Studien zum Thema Einzel- vs. Paarprogrammierung zusammen
 - Enthält einige der zuvor genannten Studien



Ergebnisse Dybå et al. 2007 Qualität

Umgekehrt
proportional
zur Varianz

Study

Effect size and 95% confidence interval

Relative
weight

P07a
S06a
S00
S03
S05b
P07b
S02
S06c
S06b
P98
S06d

Williams et al. 2000

P = Profis
S = Studenten

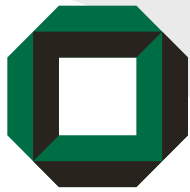
Overall effect

23.24
22.85
18.64
9.80
7.95
4.73
4.53
2.81
2.33
2.02
1.09

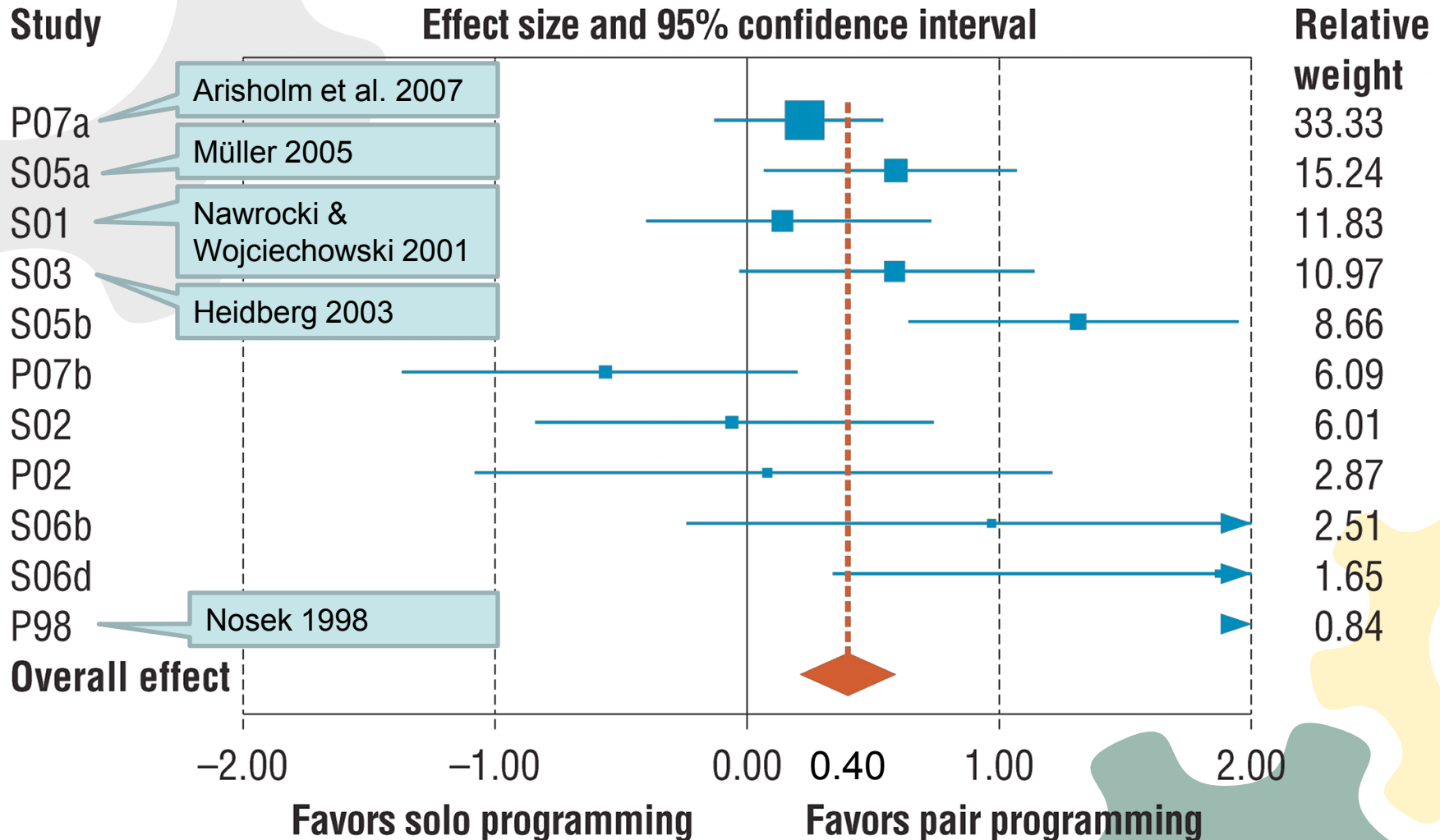
-2.00 -1.00 0.00 0.38 1.00 2.00

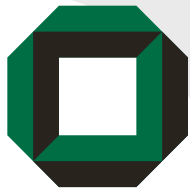
Favors solo programming

Favors pair programming



Ergebnisse Dybå et al. 2007: Dauer





Ergebnisse Dybå et al. 2007: Aufwand

Study

P07a

S05a

S05b

S05c

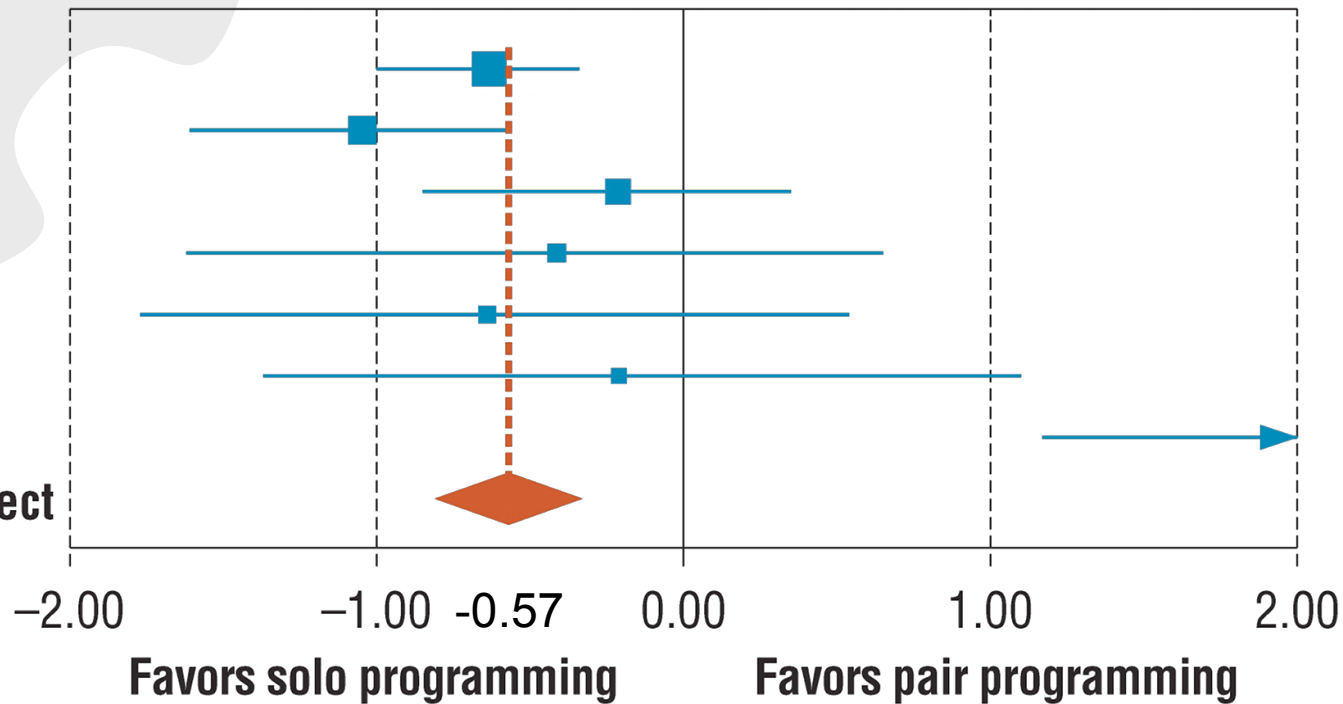
S06b

S06d

S06c

Overall effect

Effect size and 95% confidence interval



Relative weight

47.66

21.17

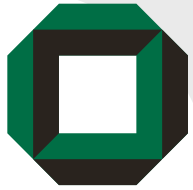
15.73

4.46

4.04

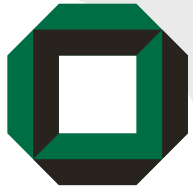
3.88

3.07



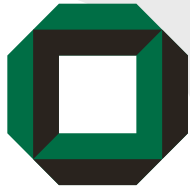
Stand Paarprogrammierung

- Programmierer im Paar müssen aufeinander abgestimmt sein
- Beste Arbeitsteilung unbekannt
- Vorteil bei Qualität und Dauer
- Nachteil beim Aufwand
- Einfache Fehler werden vom Paar schneller entdeckt als vom einzelnen. Schwierige nicht.



Erfahrungen mit testgetriebener Entwicklung

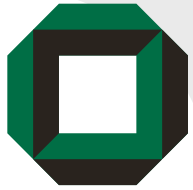
- Testgetriebene Entwicklung ist ungewohnt und schwer zu erlernen, da neue Denkweise
- Gewöhnung dauert eventuell Monate
- Studenten: Automatisches Ausführen von Tests ist optimales Verfahren
- Gefahr: Falsches Gefühl der Sicherheit
 - Durch erfolgreiche Testfälle erzeugt
 - Verhindert neue effiziente Testfälle



Quantitative Studien zur testgetriebenen Entwicklung

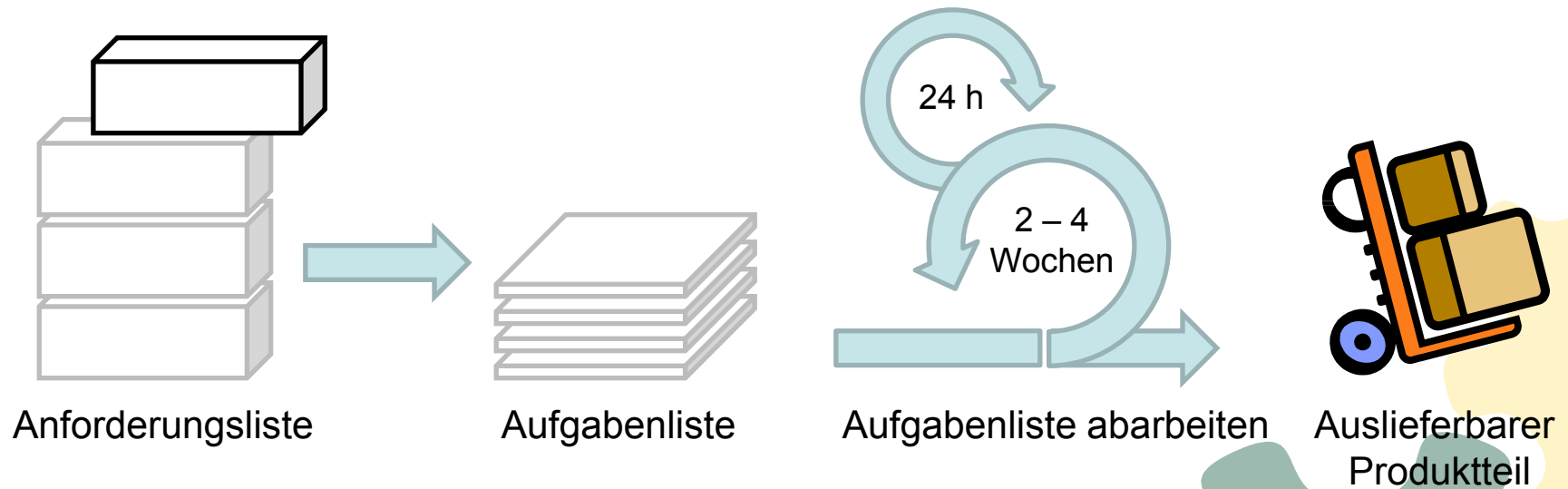
Studie	Teilnehmer	Produktivität	Qualität
TGE vs. Wasserfall-Prozess			
Müller & Hagner 2002	Studenten	schlechter	Kein Unterschied
George & Williams 2003	Profis	schlechter	besser
Edwards 2004	Studenten	Keine Angabe	besser
Canfora et al. 2006	Profis	schlechter	Kein Unterschied
TGE vs. Iterative Test Danach Entwicklung			
Pančur et al. 2003	Studenten	Kein Unterschied	Keine Angabe
Geras et al. 2004	Profis	Kein Unterschied	Keine Angabe
Erdogmus et al. 2005	Studenten	besser	Kein Unterschied

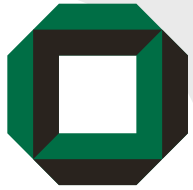
- Viele Studien mit Studenten → Übertragbarkeit?
- Insgesamt ein uneinheitliches Bild → weiterer Forschungsbedarf



Scrum

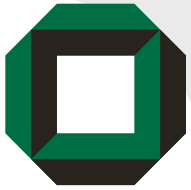
- Scrum ist ein Vorgehensmodell für das agile Projektmanagement.





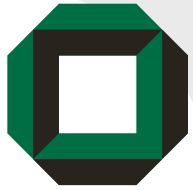
Scrum – Artefakte

- Anforderungsliste (engl. product backlog)
 - Enthält die Produkthanforderungen und eine Liste aller Projektarbeiten
 - Die Anforderungen werden vor jedem Sprint durch den Auftraggeber priorisiert
- Aufgabenliste (engl. sprint backlog)
 - Enthält alle Aufgaben, die zur Erfüllung des aktuellen Sprint-Ziels notwendig sind, inklusive einer kurzen Beschreibung.
- Restaufwands-Diagramm (engl. burndown chart)
 - Stellt den noch zu erbringenden Restaufwand für den aktuellen Sprint grafisch dar.
- Hindernisliste (engl. impediment backlog)
 - Enthält alle Hindernisse des Projektes, die der Scrum Master zusammen mit dem Team bespricht und versucht, diese auszuräumen.



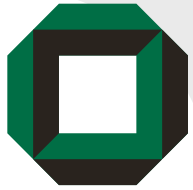
Scrum – Rollen

- Auftraggeber (engl. product owner)
 - Legt die Anforderungen an das Produkt fest sowie dessen Auslieferungstermin.
 - Stellt das Budget zur Verfügung und akzeptiert bzw. lehnt Arbeitsergebnisse ab.
 - Priorisiert Anforderungen und passt Anforderungen sowie deren Priorität für jeden Sprint an.
- Scrum Master
 - Stellt die Einhaltung der Scrum-Werte und -Techniken sicher.
 - Sorgt dafür, dass das Entwicklungsteam vollständig, funktional und produktiv ist und schützt das Entwicklungsteam vor äußeren Störungen.
 - Beseitigt Hindernisse und unterstützt die Zusammenarbeit zwischen allen Rollen.
- Entwicklungsteam
 - Besteht aus Personen mit unterschiedlichen Fachgebieten (Programmierer, UI-Designer, ...) und organisiert sich selbst.
 - Mitglieder eines Teams können sich nur zwischen Sprints ändern.



Scrum – Treffen

- Sprintplanung
 - Das Entwicklungsteam wählt die Anforderungen aus den Produktanforderungen aus, die es in diesem Sprint erledigen kann und erstellt zusammen mit dem Scrum Master die Aufgabenliste.
- Tägliches Scrum-Treffen
 - Jedes Mitglied des Entwicklungsteams beantwortet folgende Fragen:
 - Was hast Du gestern getan?
 - Was wirst Du heute tun?
 - Welche Hindernisse sind in Deinem Weg?
- Review-Treffen
 - Das Entwicklungsteam stellt die Ergebnisse des Sprints vor, z.B. in Form einer Demonstration der neuen Features.
 - Es werden keine Folien verwendet!
- Retrospektive
 - Der zurückliegende Sprint wird analysiert
 - Teilnehmer sind das Entwicklungsteam, der Scrum Master, der Auftraggeber sowie evtl. der Endkunde



Literatur

- Müller, Tichy, *Case Study: Extreme Programming in a University Environment*, ICSE, 2001, 537-544.
- Beck, *Extreme Programming Explained*, Addison Wesley, 2000
- Beck, *Extreme Programming Explained*, 2nd edition, Addison Wesley, 2005
- Martin Fowler, *Refactoring*, Addison Wesley, 1999
- Link, *Unit Tests mit Java*, dpunkt.verlag, 2002