

Testbarkeit im Entwicklungsprozess

TAE-Kolloquium "Testen im System- und Software-Lifecycle"
30. November 2005

Dr. Stefan Jungmayr
Teradyne Diagnostic Solutions

Sprecher des GI-Arbeitskreises "Testen objektorientierter Programme"
www.testbarkeit.de

TERADYNE



Assembly Test Division
Circuit Board
Inspection & Test

Semiconductor Test Division
Device & Wafer
Test

Diagnostic Solutions (DS) Division
Automotive Test
& Diagnostics

Broadband Test Division
Telecommunication
Systems Test

weltweit führender Lieferant von Diagnose- und Informationslösungen (www.teradyne.com)

Umsatz 2004 (gesamt): US\$ 1,69 Milliarden

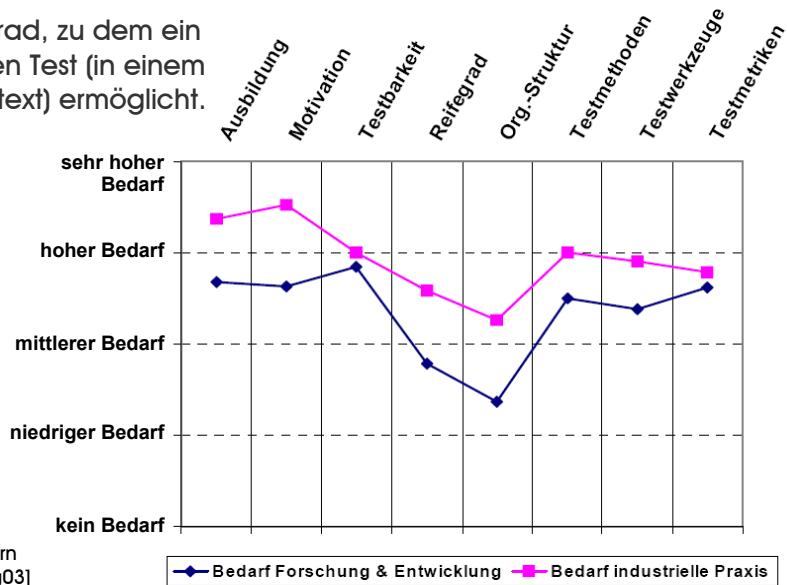
Mitarbeiter (gesamt): 5200

Mitarbeiter (DS): 390

Kunden (DS): BMW, Claas, DC, Ford, GM, Honda, Jaguar, Knorr Bremse, MAN, Saab, Volvo

Testbarkeit

Testbarkeit ist der Grad, zu dem ein Software-Artefakt den Test (in einem bestimmten Testkontext) ermöglicht.



Umfrage unter 21 Teilnehmern einer Test-Fachtagung [Jung03]

Testbarkeit im Entwicklungsprozess – TAE-Kolloquium "Testen im System- und Software-Lifecycle", 30.Nov.2005

Copyright © 2005 Dr. Stefan Jungmayr

Testprobleme ...

"Früher konnte man sicher sein, dass der Kindersitz ein abgetrenntes Modul ist und der Nebelscheinwerfer ein anderes. Seit aber der Sitz einen Einlullmotor hat, interferieren beide mit der Elektronik. Wenn jetzt das Baby Milchsäure ausspuckt, beginnen Wechselwirkungen im ganzen Auto. [...] Dann suchen Sie einmal den Fehler!" [Duec05]

Testbarkeit im Entwicklungsprozess – TAE-Kolloquium "Testen im System- und Software-Lifecycle", 30.Nov.2005

Copyright © 2005 Dr. Stefan Jungmayr

Wo stehen Sie?

- Wie hoch ist Ihr Testaufwand (in Prozent)?
 - inkl. Fehlersuche / Debugging
- Haben Sie erhöhten Testaufwand, weil die Software den Test nicht ausreichend unterstützt?
- Was tun sie gegen Testbarkeitsmängel?

Was kann man tun?

- konventionell:
 - anerkannten Entwurfsrichtlinien folgen ("guter Entwurf")
 - objektorientiert entwickeln
- testgetrieben entwickeln
- Entwurf für Testbarkeit:
 - Entwickler & Tester in testbarem Entwurf trainieren
 - Entwurfsrichtlinien definieren
 - Testbarkeit in Reviews berücksichtigen
- Motivation:
 - an das Gute im Entwickler appellieren
- ??

Implementierungs- versus Test-Sicht

Implementierung

- Aufruf von direkten Dienstleisterklassen (indirekte Dienstleisterklassen verborgen durch Information-Hiding)
- Klasse wird eingebunden in System/Rahmenwerk
- Code wird bei Bedarf angepasst
- Priorität: es läuft und stürzt nicht ab, gute Performanz
- Komplexität und Dokumentation sekundär
- Beobachtung über Debugger möglich
- Fehler-Isolation z.T. intuitiv

Test

- auch indirekte Dienstleisterklassen für Test relevant
- Klasse muss isoliert werden, Zustand soll möglichst direkt gesetzt werden
- Code kann nicht verändert werden
- Priorität: Wiederholbarkeit der Testergebnisse
- Verständnis von fremden Code erforderlich und schwierig
- Beobachtung nur über Schnittstelle möglich
- Fehlernachweis schwierig

Anforderungen von Testern

- produktbezogen:
 - angemessene Komplexität
 - isolierter Test von Komponente möglich
 - Zustand der Komponenten setzbar, beobachtbar
 - Test reproduzierbar, automatisierbar, robust, schnell
 - Fehler lässt sich Komponente zuordnen
- prozessbezogen:
 - Anforderung verfügbar, aktuell und stabil
 - Anforderungen testbar und verfolgbar

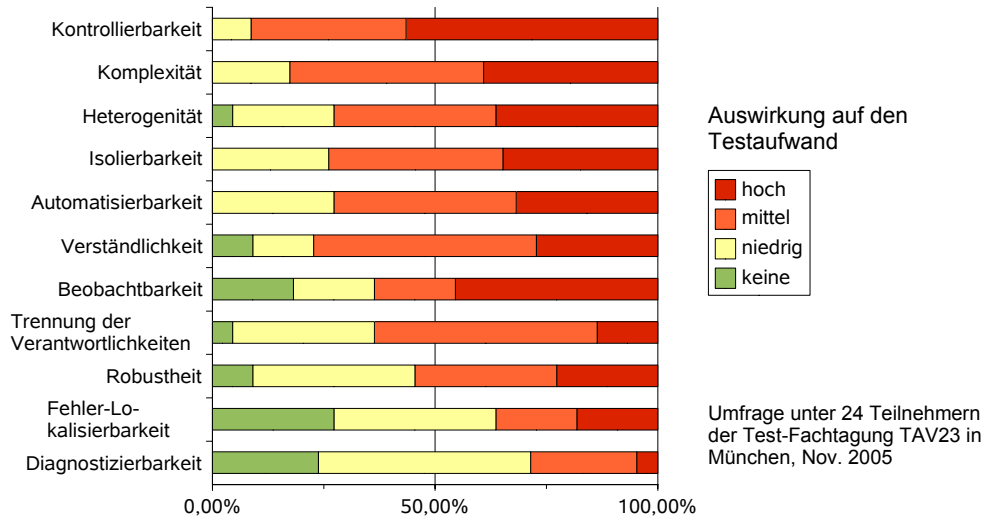
Anforderungen von Entwicklern (Wartung)

- **Unterstützung der Fehler-Isolation:**
 - Fehlermeldungen und Auslöser eindeutig
 - Konfiguration des Systems über Fehlermeldung bzw. UI nachvollziehbar
 - Test im laufenden Betrieb möglich, vernachlässigbare Wirkung auf Systemverhalten
 - Systemzustand zugänglich
 - Log-Informationen konfigurierbar
 - selbstdefinierte Abfragen an das System möglich

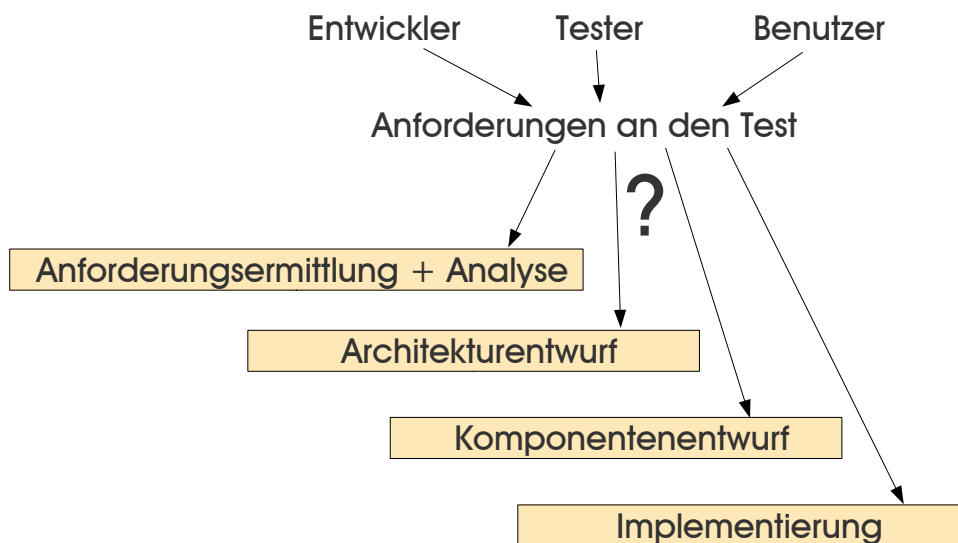
Anforderungen von Benutzern

- **Unterstützung der Fehlererkennung:**
 - Systemzustand zugänglich
 - Systemzustand & Fehlermeldungen nachvollziehbar
 - Prüfung der Integrität durch Selbst-Test
- **Unterstützung der Fehlermeldung:**
 - Fehlermeldung und Systemzustand bei Fehlereintritt dokumentierbar
 - Weiterleitung der Fehlermeldung an Entwickler automatisiert

Priorität der Anforderungen



Testbarkeitsanforderungen und Implementierung



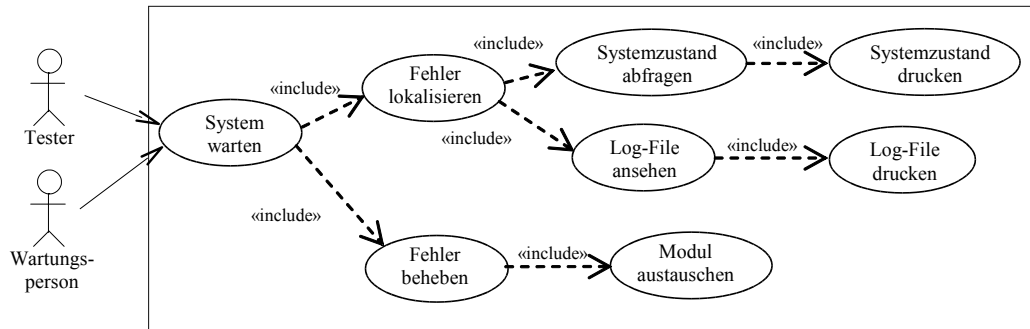
Testbarkeit in Anforderungsermittlung + Analyse

- Testbarkeit der Anforderungen erhöhen:
 - Komplexität der Anwendungsfälle reduzieren
 - Anforderungen quantitativ, ausreichend detailliert formulieren
 - schwer testbare Anforderungen identifizieren
- testkritische Domänenklassen identifizieren
 - kritische Anwendungslogik
 - schwer beobachtbar/automatisierbar
 - Zugriff auf langsame Ressourcen
- Abhängigkeiten von Anwendungsfällen:
 - Vorbedingungen lockern
 - zeitliche Abhängigkeiten reduzieren
 - untere Multiplizitätsgrenzen lockern
 - Zugriff auf gemeinsame Klassen red.
- Testbarkeitsanforderungen definieren:
funktional / nicht-funktional

Testbarkeitsanforderung – nicht-funktional (Bsp.)

- *Test in Isolation:*
"Für 90% aller Klassen erfordert JUnit-Testsetup die Instantiierung von Objekten aus weniger als 7 Dienstleisterklassen."
- *Fehler-Lokalisierbarkeit:*
"Durchschnittlich < 30 Minuten erforderlich, um einen kritischen Fehler zu isolieren."
- *Beobachtbarkeit:*
"Abstrakter Zustand testkritischer Komponenten ist über eine Testschnittstelle abrufbar."

Testbarkeitsanforderung - funktional (Bsp. 1/2)



Testbarkeitsanforderung - funktional (Bsp. 2/2)

Anwendungsfall Systemzustand abfragen

Akteur: Tester, Wartungsperson

Ablauf: Der Akteur fragt den Systemzustand ab. Der Systemzustand beinhaltet Ereignisse (Ein- und Ausgabe-Operationen, Fehlerereignisse) und Systemkennzahlen (CPU-Nutzung, Speichernutzung, Anzahl der Objektinstanzen, Anzahl der Threads). Die Abfrage des Systemzustands ist passwortgeschützt. Der Akteur kann die Ausgaben sortieren und filtern (nach Zeitpunkt und Kritikalität). Der Akteur kann Ereignisse und Systemkennzahlen auswählen und drucken (**include** „Systemzustand drucken“).

Vorbedingung: keine (soweit möglich, soll der Anwendungsfall in allen möglichen Systemzuständen ablauffähig sein)

Nachbedingung: Der aktuelle Systemzustand wurde ausgegeben.

Ausnahmefälle: keine

Testbarkeit im Architekturentwurf

- Komplexität reduzieren
- Austauschbarkeit sicherstellen
- Verantwortlichkeiten trennen
- Beobachtbarkeit sicherstellen
- Automatisierbarkeit sicherstellen
- Fokus auf testkritische Klassen

- Mock-Objekte für systemweite Dienste (Mock-Plattform [Völt05])
- Objektfabriken
- Dependency Injection
- Singleton-Muster vermeiden
- "Durchstich" für Test in Isolation

- Testschnittstelle
- abstrakter Zustand abfragbar
- Zugriff auf Laufzeitumgebung: Anzahl Objekte / Threads / Speicherverbrauch
- Selbstauskunft über Version und Konfiguration
- Logging auf unterschiedlichen Leveln (tw. Kundenanforderung)

Testbarkeit im Komponententwurf (Beispiele)

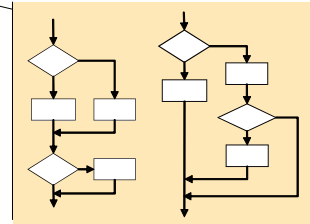
- Komplexität reduzieren
 - hohe Schnittstellen-Komplexität vermeiden
 - tiefe Vererbungshierarchien vermeiden
 - zyklische Klassenabhängigkeiten vermeiden
 - Abh. zu indirekten Dienstleisterklassen vermeiden
- Fehler-Lokalisierbarkeit verbessern
 - Yo-Yo Effekt reduzieren
 - implizite Abhängigkeiten (Seiteneffekte) vermeiden
- Fehler-Lokalität verbessern
 - defensive Programmierung an Subsystemgrenzen

Testbarkeit in der Implementierung (Beispiele)

- **Verständlichkeit sicherstellen**
 - “elegante”, trickreiche Lösungen vermeiden
 - implizite Kontrolllogik vermeiden
- **Kontrollierbarkeit sicherstellen**
 - komplexe Schleifenkonstrukte vermeiden
 - Rekursion vermeiden
 - unerreichbare Pfade vermeiden
 - unerreichbare Ausgabewerte vermeiden

• erst auf Performanz optimieren, wenn Lösung korrekt und testbar

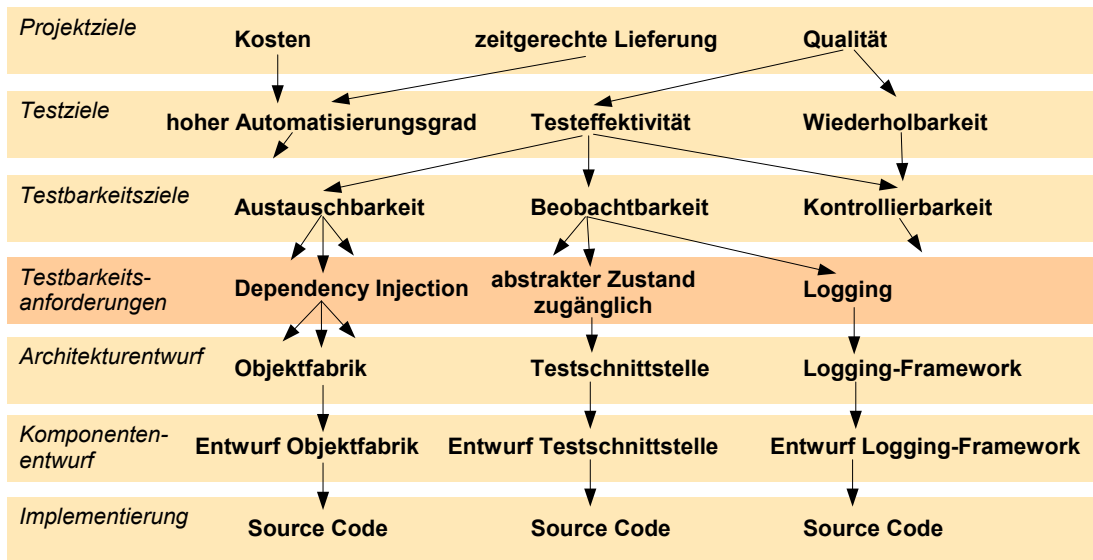
• Trennung von Iterator-Code und “Arbeits-Code”.



Testbarkeit bewerten

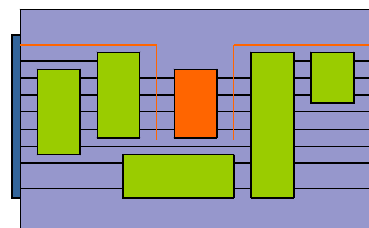
- **perspektiven-basierte Reviews**
 - Testfälle erstellen
- **Durchstich mit Test**
- **Komplexitätsmetriken**
 - Schnittstellen-Metriken [Snee]
 - Kontrollfluss-Metriken: McCabe
 - Abhängigkeits-Metriken: ACD [Lakos96]
- **Metriken zur Austauschbarkeit**
 - fest-verdrahtete Abhängigkeiten [Jung04]

Verfolgbarkeit der Testbarkeitsanforderungen



Vergleich: Hardware-Testbarkeit

- VLSI:
 - Anforderung: jedes IC muss getestet werden
 - Problem: Mangel an Steuerbarkeit / Beobachtbarkeit
 - Lösung: Boundary-Scan-Architektur (IEEE 1149.1)
- Elektronische Ausrüstung:
 - MIL-STD-2165 (1985)
 - u.a.: Testbarkeitsanforderungen
- U-Boot-Systeme:
 - Testability Analysis Handbook (1992)
 - u.a.: Testbarkeitsanforderungen



Ansätze für mehr Software-Testbarkeit

- Ad-hoc-Ansatz
- testgetriebene Entwicklung
- anforderungsgetriebene (Testbarkeits-) Entwicklung

Vorteile:

- Einbindung in Entwicklungsprozess (V-Modell, RUP, etc.)
- Verfolgbarkeit (Traceability)
- Wiederholbarkeit
- Effektivität

Was kann man tun?

- "guter Entwurf", objektorientiert entwerfen
- ~~an das Gute im Entwickler appellieren~~
Testbarkeitsanforderungen definieren
- testkritische Klassen identifizieren
- Durchstich und Test / testgetrieben entwickeln
- Testprobleme als eigene Kategorie in Fehler-DB
- Entwurfsrichtlinien definieren & prüfen (z.B. Reviews)
- Entwickler & Tester bzgl. Testbarkeit und Testbarkeitsanforderungen trainieren

Jene, die Testbarkeits-Technologien beherrschen und anwenden, werden einen wesentlichen Wettbewerbsvorteil für sich verbuchen können.

- Testbarkeitsanforderungen in Entwicklungsprozessen und -Produkten verankert.
- Know-How aufgebaut:
 - **Analytiker**: kennen konstruktive Möglichkeiten, die Testen erleichtern → können Testbarkeitsanforderungen definieren
 - **Entwickler**: können Testbarkeitsanforderungen umsetzen
 - **Projektleiter**: treffen richtige Trade-Offs zwischen Testbarkeit und anderen Anforderungen / Test- und Entwicklungskosten
- Testbarkeit im Entwicklungsprozess kontrolliert.
- Kritische Testbarkeitsmängel werden vermieden bzw. frühzeitig erkannt und behoben.