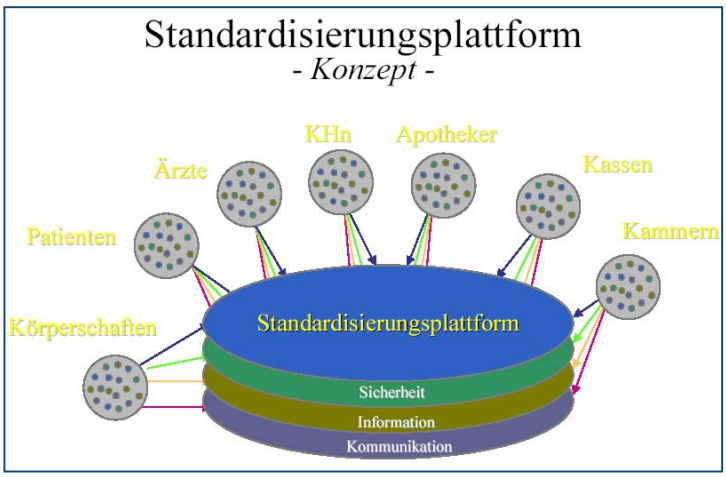
Überblick

- Komponenten-Architekturen
  - Enterprise Java Beans (EJB)
  - Corba Component Model (CCM)
- Komponenten-Spezifikation
- Fachliche Validierung von Komponenten
- Vorgehensweise

Health-Care Plattformen

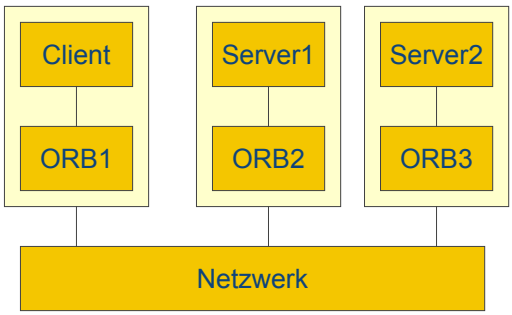


CSVHC04 Folie 3 Anforderungen an Komponentenspezifikationen Prof. Dr. Mario Winter, FH Köln

Verteilte, heterogene Dienste und Systeme

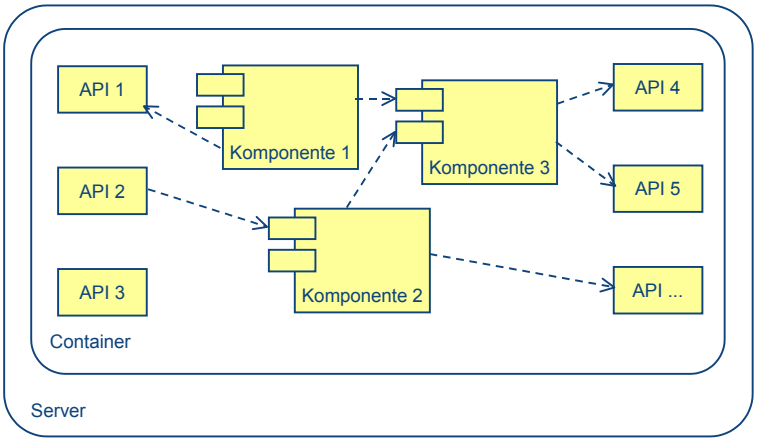
- Welche Dienste werden angeboten?
- Was genau kann der Client erwarten?

- Technische Problemstellungen:
- Nebenläufigkeit
  - Nondeterminismus
- Getrennte Adressräume
  - Beobachtbarkeit
- Ortstransparenz gefordert!
  - Zugriff auf entfernte Daten
  - Verschieben von Daten
- Performanz?
- Skalierbarkeit?
- Sicherheit?
  - Datenschutz?
  - Ausfallsicherheit?



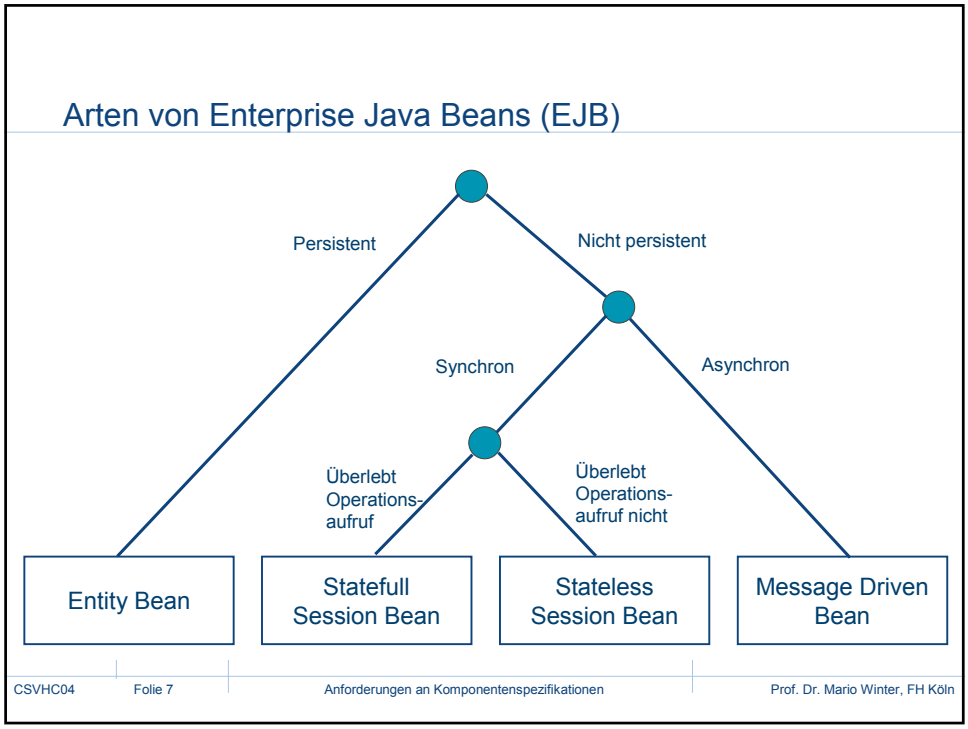
CSVHC04 Folie 4 Anforderungen an Komponentenspezifikationen Prof. Dr. Mario Winter, FH Köln

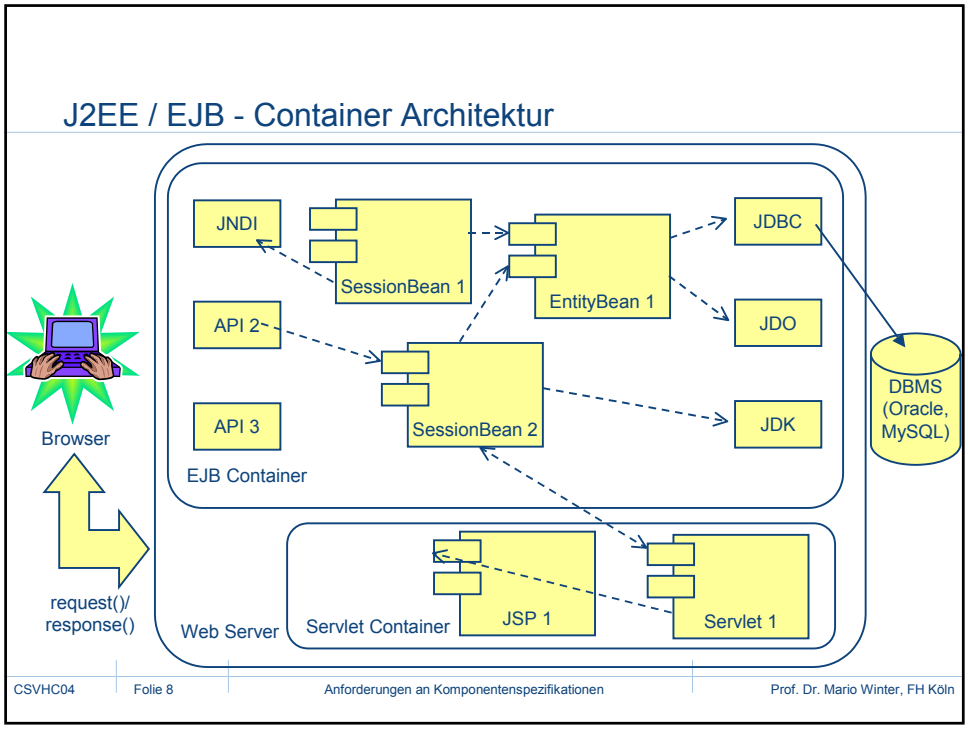
Die Antwort der Industrie: Komponenten-Programmiermodell „Container“



J2EE – Enterprise Java Beans

- Java 2 Enterprise Edition
- Idee: Business Logik für Web-Applikationen
- Enterprise Java Beans (EJB) ist nur ein API in J2EE
- EJB Container übernimmt
  - Persistenz (Datenbank-Zugriffe)
  - Transaktionen / Mehrbenutzerbetrieb
  - Sicherheit
  - Load-Balancing
  - ... (Container-Provider-abhängig)
- Zusammenwirken mit anderen Web-Technologien
  - JSP
  - Servlets
  - Applets
  - ...


Arten von Enterprise Java Beans (EJB)
Persistent
Nicht persistent
Synchron
Asynchron
Überlebt Operations-aufruf
Überlebt Operations-aufruf nicht
Entity Bean
Statefull Session Bean
Stateless Session Bean
Message Driven Bean
CSVHC04
Folie 7
Anforderungen an Komponentenspezifikationen
Prof. Dr. Mario Winter, FH Köln

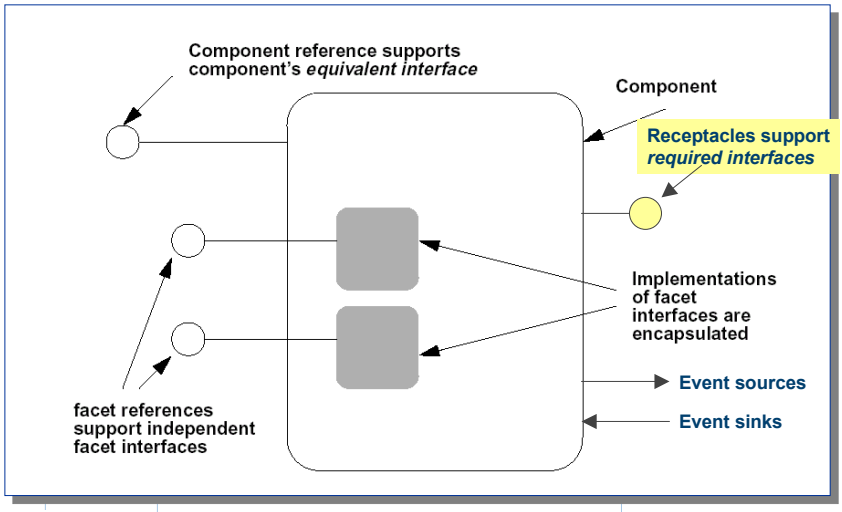

J2EE / EJB - Container Architektur
JNDI
SessionBean 1
EntityBean 1
JDBC
API 2
JDO
SessionBean 2
API 3
JDK
DBMS (Oracle, MySQL)
EJB Container
Browser
request()/ response()
Web Server
Servlet Container
JSP 1
Servlet 1
CSVHC04
Folie 8
Anforderungen an Komponentenspezifikationen
Prof. Dr. Mario Winter, FH Köln

Zugriff auf Enterprise-Beans

- Remote-Interface
  - Definiert die Schnittstelle der Bean mit den fachlichen Operationen, also alle für die „Außenwelt“ interessanten Operationen
  - Erweitert `javax.ejb.EJBObject`
- Home-Interface
  - Definiert die Lebenszyklus-Operationen, über welche die Bean erzeugt, gefunden und zerstört werden kann
  - Erweitert `javax.ejb.EJBHome` (und damit `Java.rmi.remote`)
- Bean-Klasse
  - Realisiert die fachlichen Operationen der Bean
  - Entity Beans implementieren `javax.ejb.EntityBean`
  - Session Beans implementieren `javax.ejb.SessionBean`
  - Message Driven Beans implementieren `javax.ejb.MessageDrivenBean`
- Primärschlüssel (Nur Entity Beans)
  - Stellen Schlüssel der Datenbank zur Verfügung
  - Implementiert `javax.io.Serializable`
- Frage: Welche anderen Komponenten/Dienste benötigt eine EJB???

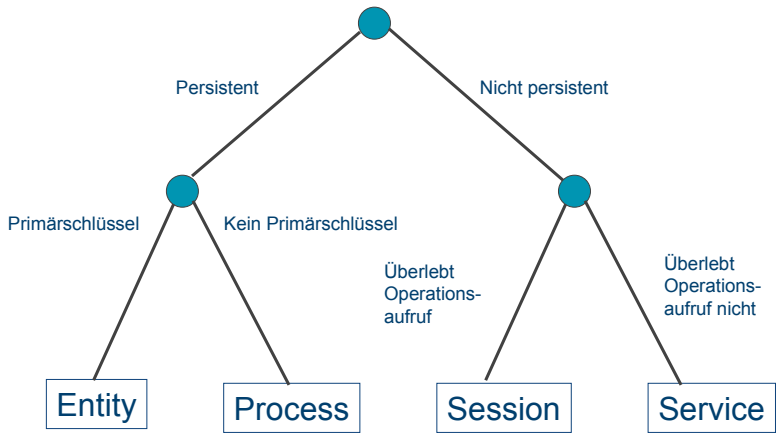
CSVHC04 Folie 9 Anforderungen an Komponentenspezifikationen Prof. Dr. Mario Winter, FH Köln

Antwort: Corba Component Model (CCM)



CSVHC04 Folie 10 Anforderungen an Komponentenspezifikationen Prof. Dr. Mario Winter, FH Köln

CCM Komponentenkategorien



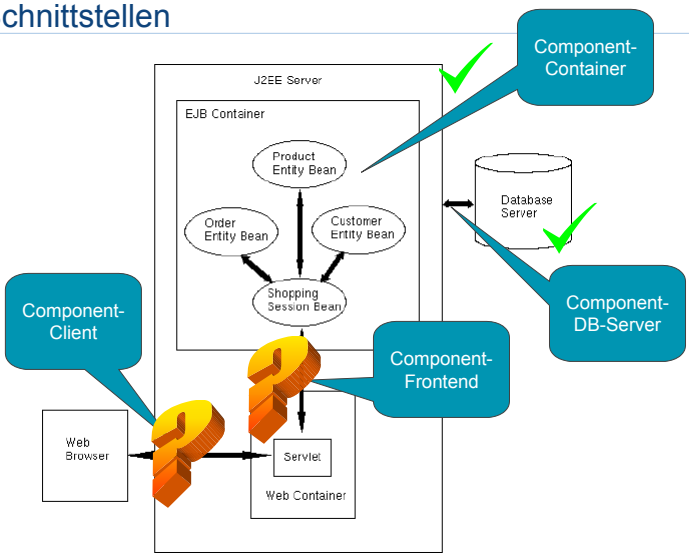
Komponenten- Architekturen: Zusammenfassung

- EJB
  - 4 Kategorien: Entity / (statefull/stateless) Session / Message Driven
  - Home / Remote – Interfaces
  - XML-Deployment-Deskriptor
- CCM
  - 4 Kategorien: Entity / Process / Session / Service
  - Home / Equivalent / Facet / Receptacle – Interfaces
  - Event-Source / -Sink
  - IDL / CIDL Spezifikation
- Spezifikation eher auf syntaktischer Ebene
  - Java/IDL-Interfaces beschreiben Typen
  - In EJB nur angebotene (ensured) Interfaces betrachtet
  - Z.T. Lebenszyklus vorgegeben (Zustandsdiagramm)
  - Nur technische Operationen vorgegeben
- Hauptproblem: Fachliche Operationen bleiben außer acht!

Wo sind wir?

- Komponenten-Architekturen
  - EJB
  - CCM
- **Komponenten-Spezifikation**
- Fachliche Validierung von Komponenten
- Vorgehensweise

EJB: Schnittstellen



Komponenten-Spezifikation (Design by Contract)

- Dienstnutzer vs. Dienstleister
  - Vorbedingung
  - Nachbedingung
  - Invarianten
- Vertrag
  - Wenn der Dienstnutzer die Vorbedingung erfüllt, dann garantiert der Dienstleister die Nachbedingung!
  - Dienstnutzer **und** Dienstleister erfüllen ihre Invarianten vor und nach jeder Methodenausführung!

CSVHC04 Folie 15 Anforderungen an Komponentenspezifikationen Prof. Dr. Mario Winter, FH Köln

Spezifikation fachlicher Operationen mit Zusicherungen

- Vorbedingung
  - Definiert, welche Voraussetzungen vor dem Aufruf einer Operation gelten müssen, damit die (durch die Nachbedingung) definierte Leistung erbracht werden kann
  - Bezieht sich auf Eingabe-Parameter der Operation und Zustand der Komponente
- Nachbedingung
  - Definiert das durch das dienstleistende Objekt abgelieferte Ergebnis der Operation unter der Annahme, dass die Vorbedingung beim Aufruf erfüllt war
  - Bezieht sich auf Ausgabe-Parameter der Operation und Zustand der Komponente
- Komponenteninvariante
  - Erlaubt es, gemeinsame Teile aus allen Vor- und Nachbedingungen der Operationen der Komponente herauszuziehen und nur einmal zu notieren
  - Gilt immer vor und nach der Ausführung von öffentlich sichtbaren Operationen
  - Bezieht sich auf Zustand der Komponente

CSVHC04 Folie 16 Anforderungen an Komponentenspezifikationen Prof. Dr. Mario Winter, FH Köln

Beispiel: Stapel (Stack) mit begrenzter Kapazität

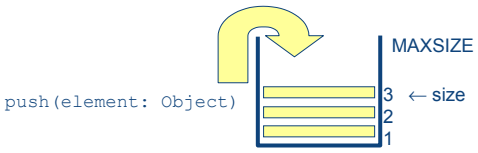
Klasse `BoundedStack`

Zustandserhaltende Operationen

```
size():integer;      // Anzahl gestapelter Elemente
MAXSIZE(): integer; // Maximale Anzahl
top():Object;        // Liefert Zeiger auf oberstes Element
```

Zustandsverändernde Operationen

```
BoundedStack(maxSize: integer);// Konstruktor
~BoundedStack();               // Destruktor
push(element: Object);         // Stapelt Element
pop();                         // Entfernt oberstes Element
```



Beispiel: Spezifikation der Komponente BoundedStack

```
component BoundedStack {
        /** invariant@ this.size() >= 0 AND this.size() <= this.MAXSIZE */
        public BoundedStack (Integer maxSize){
        /** pre@  maxSize > 0 */
        /** post@ self.MAXSIZE = maxSize@pre */
        }
        public void push (Object item) throws FullStackException {
        /** pre@ this.size() < this.MAXSIZE */
        /** post@ this.size() = this.size()@pre + 1 */
        }
        public Object top () throws EmptyStackException {
        /** pre@ this.size() > 0 */
        /** post@ return != null */
        }
        public void pop () throws EmptyStackException {
        /** pre@ this.size() > 0 */
        /** post@ this.size() = this.size()@pre - 1 */
        }
        public Collection all () {
        /** pre@ true */
        /** post@ (this.size() > 0 -> return.size() = this.size()) AND
                   (this.size() = 0 -> return = null)*/
        }
}
```

Wo sind wir?

- Komponenten-Architekturen
  - EJB
  - CCM
- Komponenten-Spezifikation
- **Fachliche Validierung von Komponenten**
- Vorgehensweise

Black-Box-Test der Komponentenschnittstellen

- Ziel
  - Prüfung der Schnittstellen einer Komponente, also der öffentlich sichtbaren Operationen (und Instanzvariablen)
- Nutzen
  - 1. Konformanz der Operationen einer Komponente zu ihrer Spezifikation
  - 2. Prüfung des Zusammenspiels mehrerer Operationen
  - 3. Erzeugung der für die Operationen möglichen Ausnahmen (exceptions)
- Voraussetzung
  - Vorliegen von Vor- und Nachbedingungen für jede Operation der zu testenden Komponente (Komponente Unter Test, KUT) sowie der Komponenteninvariante (Zusicherungen, Assertions)
- Ergebnis
  - Komponente, deren öffentlich sichtbare Operationen gegen die Spezifikation (Verträge) getestet sind

Aufstellen der Bedingungstabelle

- Vor- und Nachbedingungen und Invariante betrachten
- Bedingungen in ihre atomaren Prädikate zerlegen
  - Regeln der Aussagenlogik (Distributiv- und Kommutativgesetz, Absorptionsgesetz)
  - Bedingungen auf Redundanz untersuchen, um den Testaufwand zu minimieren
- Hilfreiche Fragen zur Vereinfachung:
  - Welches sind die elementaren Bedingungen der Bedingungen?
  - Bei GeneralisierungsHierarchien: Gibt es Verträge der Ober-Komponente? Wenn ja, ist die Invariante der Ober-Komponente mit UND, die Vorbedingung der Operationen der Ober-Komponente mit ODER und die Nachbedingung der Operationen der Unter-Komponente mit UND mit den entsprechenden Elementen der Unter-Komponente zu verknüpfen.
  - Können gemeinsame Prädikate aller Vor- und Nachbedingungen aller Operationen der Schnittstelle der KUT in die Komponenteninvariante ausgelagert werden?

Beispiel: Bedingungstabelle der Komponente BoundedStack

Context BoundedStack		. . .	
invariant@ self.size() >= 0 AND			
self.size() <= self.MAXSIZE			
BoundedStack() pre@ maxSize > 0			
BoundedStack()post@ self.size() = 0 AND			
self.MAXSIZE = maxSize			
push() pre@ self.size() < self.MAXSIZE			
push() post@ self.size()=self.size()@pre+1			
top() pre@ self.size() > 0			
top() post@ return != null			
pop() pre@ self.size() > 0			
pop() @post self.size() = self.size()@pre – 1			
all() pre@ true			
all() post@ self.size() > 0 AND			
(return.size() = self.size() OR			
not self.size() > 0 AND			
return = null)			

Ableitung der Konformanztestfälle

- Ziel: mit gezielten Botschaften bzw. Botschaftsfolgen Komponenteninvarianten oder Nachbedingungen bei erfüllter Vorbedingung des Anwendungsfalls zu FALSE auszuwerten
  ⇒ Fehler in der Operation gefunden
- Hierzu die Terme der Bedingungen systematisch mit Wahrheitswerten belegen:
  - **Einfache Bedingungsüberdeckung** (C_2-Test)
  - **Mehrfach-Bedingungsüberdeckung** (C_3-Test)
  - **Minimale Mehrfach-Bedingungsüberdeckung**
    Bei zusammengesetzten Termen werden alle Belegungen geprüft, bei denen die Änderung des Wahrheitswerts eines atomaren Prädikats den Wahrheitswert des gesamten Terms ändert

Konformanztestfälle aus Vorbedingungen

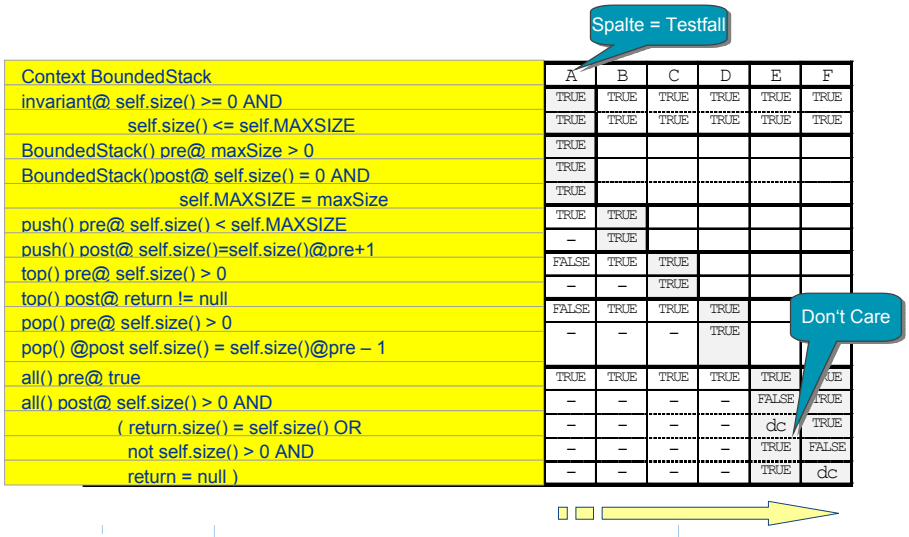
- Minimale Mehrfach-Bedingungsüberdeckung eingeschränkt auf insgesamt erfüllte Bedingungen!
  - Besteht die Vorbedingung aus einem einzigen Term A, erzeugt man einen Testfall (**true**), liegt ein negierter Term NICHT A vor, erzeugt man einen Testfall (**false**)
  - Im Falle von ODER-verknüpften Termen A und B muss jeder Term einmal den Wert **true** und einmal den Wert **false** erhalten, man erzeugt also für eine Vorbedingung vom Typ A ODER B zwei Testfälle, welche die Belegungen (**true**, **false**) und (**false**, **true**) erzwingen
  - Bei UND-verknüpfter Terme braucht nur einen Testfall mit (**true**, **true**) erzeugt werden
  - Im Falle einer Implikation WENN A DANN B zwei Testfälle mit (**false**, **false**) und (**true**, **true**) (Implikation ist wahr, wenn Prämisse nicht erfüllt)

A	B	$A \cap B$
0	0	0
0	1	0
1	0	0
1	1	1

A	B	$A \cup B$
0	0	0
0	1	1
1	0	1
1	1	1

A	B	$A \rightarrow B$
0	0	1
0	1	1
1	0	0
1	1	1

Konformanztestfälle zur Komponente BoundedStack



	A	B	C	D	E	F
Context BoundedStack						
invariant@ self.size() >= 0 AND	TRUE	TRUE	TRUE	TRUE	TRUE	TRUE
self.size() <= self.MAXSIZE	TRUE	TRUE	TRUE	TRUE	TRUE	TRUE
BoundedStack() pre@ maxSize > 0	TRUE					
BoundedStack()post@ self.size() = 0 AND	TRUE					
self.MAXSIZE = maxSize	TRUE					
push() pre@ self.size() < self.MAXSIZE	TRUE	TRUE				
push() post@ self.size()=self.size()@pre+1	–	TRUE				
top() pre@ self.size() > 0	FALSE	TRUE	TRUE			
top() post@ return != null	–	–	TRUE			
pop() pre@ self.size() > 0	FALSE	TRUE	TRUE	TRUE		
pop() @post self.size() = self.size()@pre – 1	–	–	–	TRUE		
all() pre@ true	TRUE	TRUE	TRUE	TRUE	TRUE	TRUE
all() post@ self.size() > 0 AND	–	–	–	–	FALSE	TRUE
(return.size() = self.size() OR	–	–	–	–	dc	TRUE
not self.size() > 0 AND	–	–	–	–	TRUE	FALSE
return = null)	–	–	–	–	TRUE	dc

Robustheitstestfälle

- Mit gezielten Verletzungen der Vorbedingungen einer Operation oder durch Manipulation der Testumgebung die spezifizierten Ausnahmen erzwingen
  - **Ausnahmeaktivierung**: Ausnahmen durch Zustand der Komponente und der Parameterwerte eines Operationsaufrufs erzwingen
    - Bedingungstabelle für alle ausnahmeerzeugenden Operationen `z()` betrachten
    - Vorbedingung von z() nach Testfall Y (grau schattierter Bereich) verletzt (false) ⇒ Robustheits-Testfall: zunächst Y "ausführen", dann `z()` aufrufen
    - Erwartetes Ergebnis: entsprechende Ausnahme wurde geworfen
  - **Ausnahmeinjektion**: Ausnahmen durch direkte Manipulation der Anwendungs-software oder der Umgebung des Operationsaufrufs erzwingen
    - Ablauf unter einem Debugger, absichtliche Änderung des Zustands oder des Codes
    - Manipulation der Hardware z.B. durch Vollschreiben eines Speichermediums, Ausschalten eines Gerätes o.Ä.
    - Veränderung von Datei- oder Datenbanknamen und -inhalten
    - Ausnahmesimulation, bei der durch Wrappen bestimmter Operationen die Erzeugung einer Ausnahme in der zugefügten (Test-)Software erfolgt

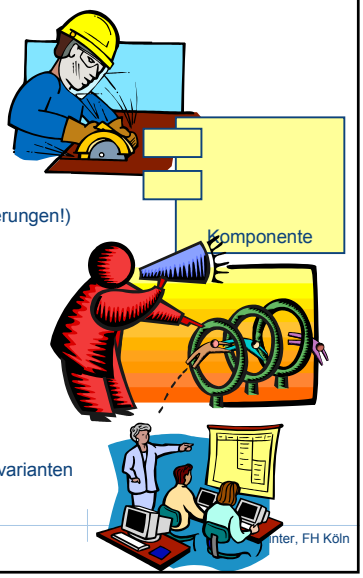
Wo sind wir?

- Komponenten-Standards
  - EJB
  - CCM
  - COM
- Komponenten-Spezifikation
- Fachliche Validierung von Komponenten
- **Vorgehensweise**

Vorgehensweise

Zwei Sichten:

- Hersteller
  - Erstellt Spezifikation (API + Zusicherungen!)
  - Liefert seine Black-Box-Testfälle mit
  - Erbringt White-Box-Überdeckungsnachweise (C1)
  - Erbringt Standard-Konformanznachweise (Container-Hersteller / Versionen / ...)
- Anwender
  - Verlangt Komponenten-Spezifikation (API + Zusicherungen!)
  - Verlangt Black-Box-Testfälle (jUnit, ...)
  - Verlangt White-Box-Überdeckungsnachweise (C1)
  - Erstellt eigene Spezifikation
  - Erstellt eigene Black-Box-Testfälle
- Infrastruktur standardisieren
  - Container-Hersteller / Versionen ...
  - Webserver-Hersteller / Versionen ...
- Mitarbeiter schulen
  - Komponentenmodellierung
  - Spezifikation mit Vor- und Nachbedingungen und Invarianten
  - Black-Box-Testverfahren



Zusammenfassung

- Komponenten-Architekturen
  - Enterprise Java Beans (EJB)
  - Corba Component Model (CCM)
  - Unzureichende Spezifikationstechniken!
  - Nur technische Aspekte betrachtet, Fachlichkeit bleibt außer acht!
- Komponenten-Spezifikation
  - Required / Ensured Interfaces spezifizieren
  - Design by Contract verwenden
  - Vorbedingungen / Nachbedingungen / Invarianten / Ausnahmen
- Black-Box Test von Komponenten
  - Konformanztests: Bedingungen erfüllt
  - Robustheitstests: Bedingungen nicht erfüllt
- Vorgehensweise
  - Hersteller- und Anwendersicht
  - Spezifikation und Testfälle verlangen

Literatur

- Booch, G., Rumbaugh, J. und Jacobson, I.: The Unified Modeling Language, Addison-Wesley, 1999 (Kapitel 9)
- Object Management Group, CORBA 3.0, CCM FTF Draft ptc/99-10-04, 1999
- Riedemann, E.: Testmethoden für sequentielle und nebenläufige Systeme, Teubner, Stuttgart, 1997
- Harry Sneed, Mario Winter: Testen objektorientierter Software – Das Praxishandbuch für den Test objektorientierter Client/Server-Systeme. Hanser Verlag, München, 2002
- Sun Microsystems, Enterprise JavaBeans™ Specification, Version 2.0, 2000