Technische Fakultät
Universität Bielefeld

Vorlesung

Softwaretest und -optimierung

Version 2012

Dr. Carsten Gnörlich

Rechnerbetriebsgruppe

Kap. 7 - Testen „im Großen“

(= Kap. 13 aus Riedemann)

Monolithisches Testen

Bisher:

- dynamisches Testen: (durch Ausführen des Programms)
- statisches Testen: (Testen ohne tatsächliche Ausführung)

→„alles auf einmal testen“-Vorgehen

Monolithischer Test führt ab gewisser Größe zu Problemen

→ siehe nachfolgende Abschnitte

→Aufteilen des Systems in Komponenten (Module) und Teilsysteme

Grenzen des monolithischen Testens

Beispiel für große Programme: Office-Pakete, SAP, ...

- zu viele Testfälle für 100%-Überdeckung
- Weg von Eingabe zur Ausgabe ist sehr lang
- Lokalisation des Defektes wird schwierig
- konkreter Einsatz nutzt nur Teil eines wiederverwendbaren Moduls

Hinweis: Unix-Kommandozeilen-Philosophie

- viele kleine Programme, die *eine* Aufgabe lösen

→ frühes Vorbild für gute Testbarkeit

Ziele des Testens

i) Funktionalität

ii) Schnittstellen

klassische Aufgaben dynam./statischen Testens

iii) Leistung / Streßtest

iv) Mengengerüst

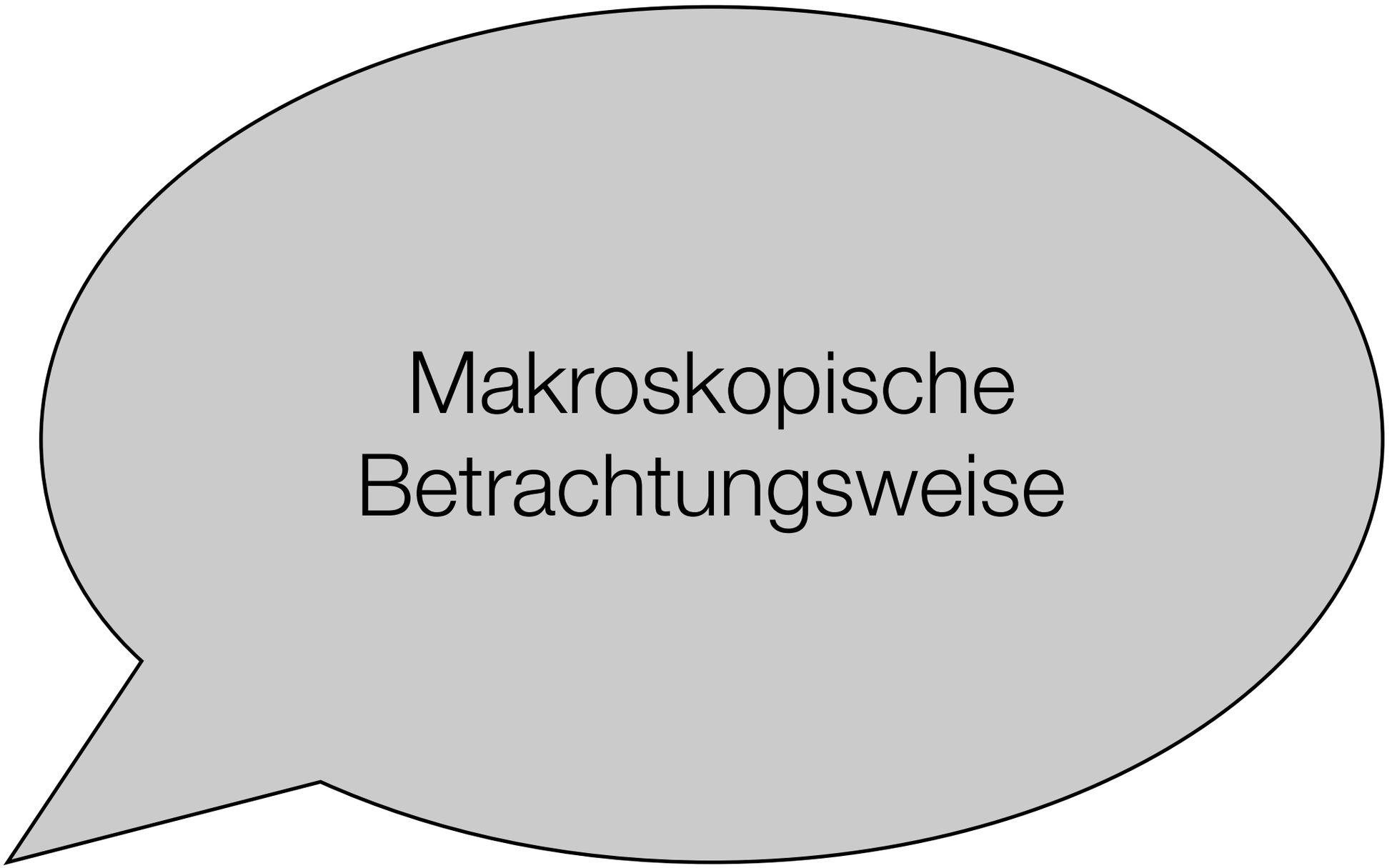
v) Verfügbarkeit

vi) Sicherheit

vii) Konfiguration

viii) Kompatibilität

ix) Benutzungsfreundlichkeit


Makroskopische
Betrachtungsweise

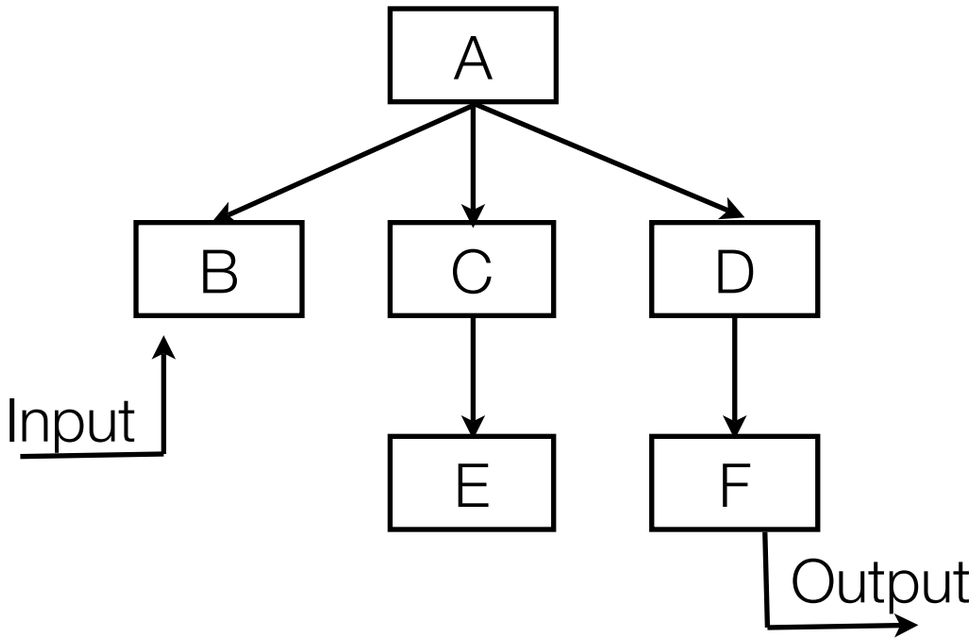
Teile des Systems als Einheit betrachten

- Module
  - Menge von Prozeduren, die unter einem Namen aufgerufen werden
  - Modul kann nur durch „Aufruf“ benutzt werden
    (Datenkapselung, Black box-Prinzip)

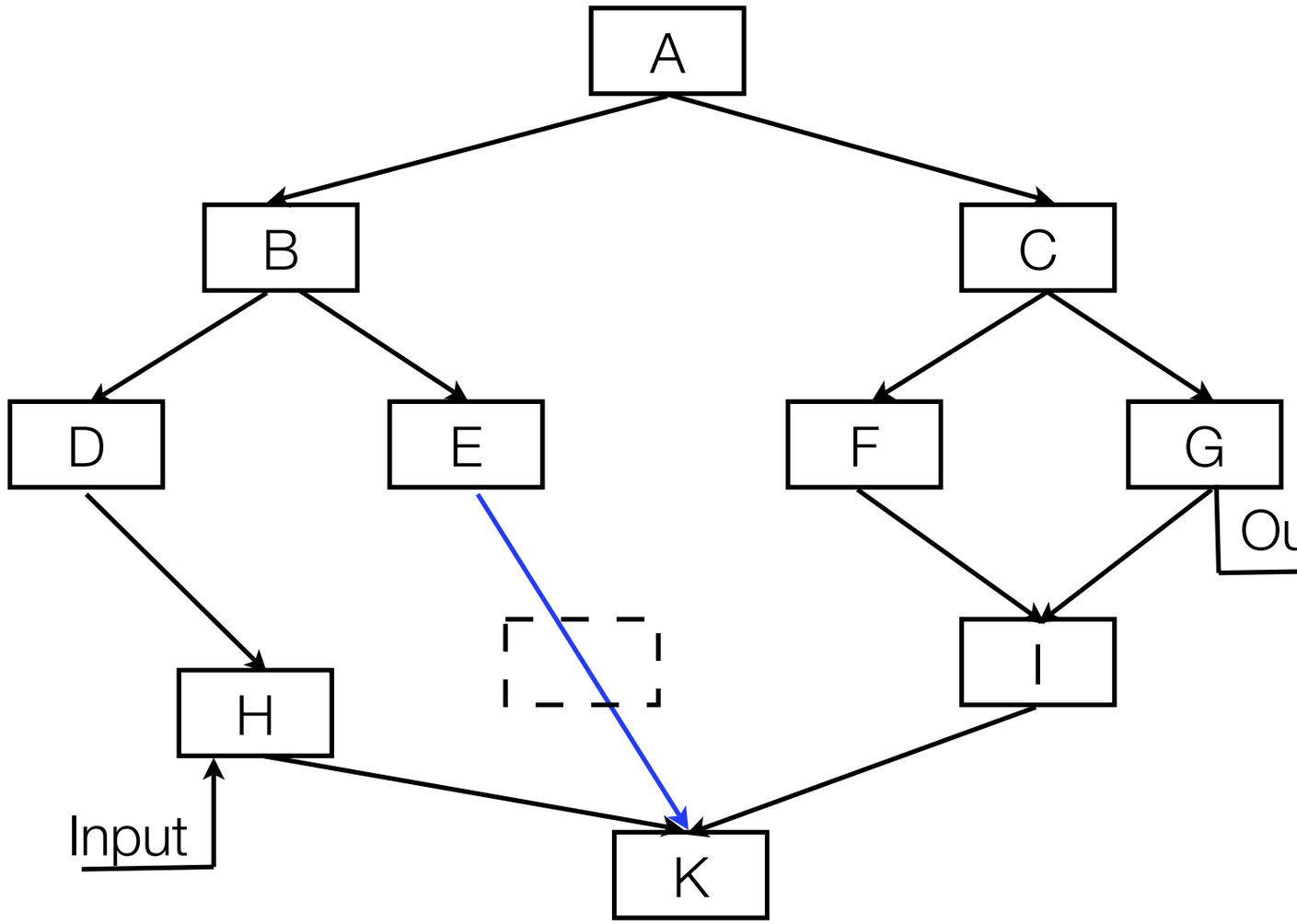
    →Testen analog zum Prozedur-/Funktionstest

- (Teil-)systeme, die aus mehreren Modulen bestehen

Benutzt-Hierarchie von Modulen



- Baum
- schichtweise aufgebaut



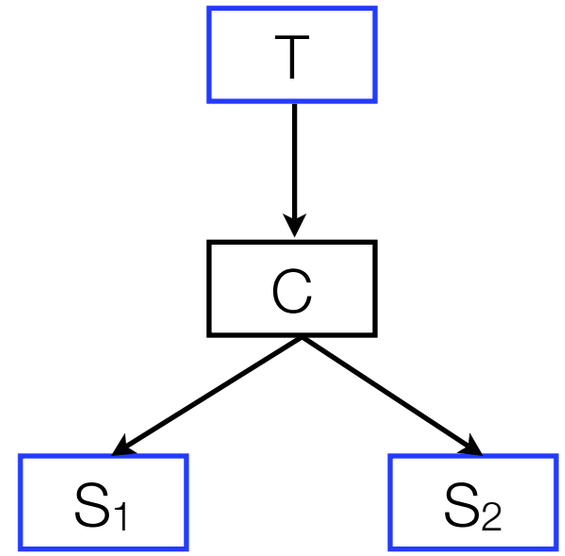
- Graph
- keine Schichtung

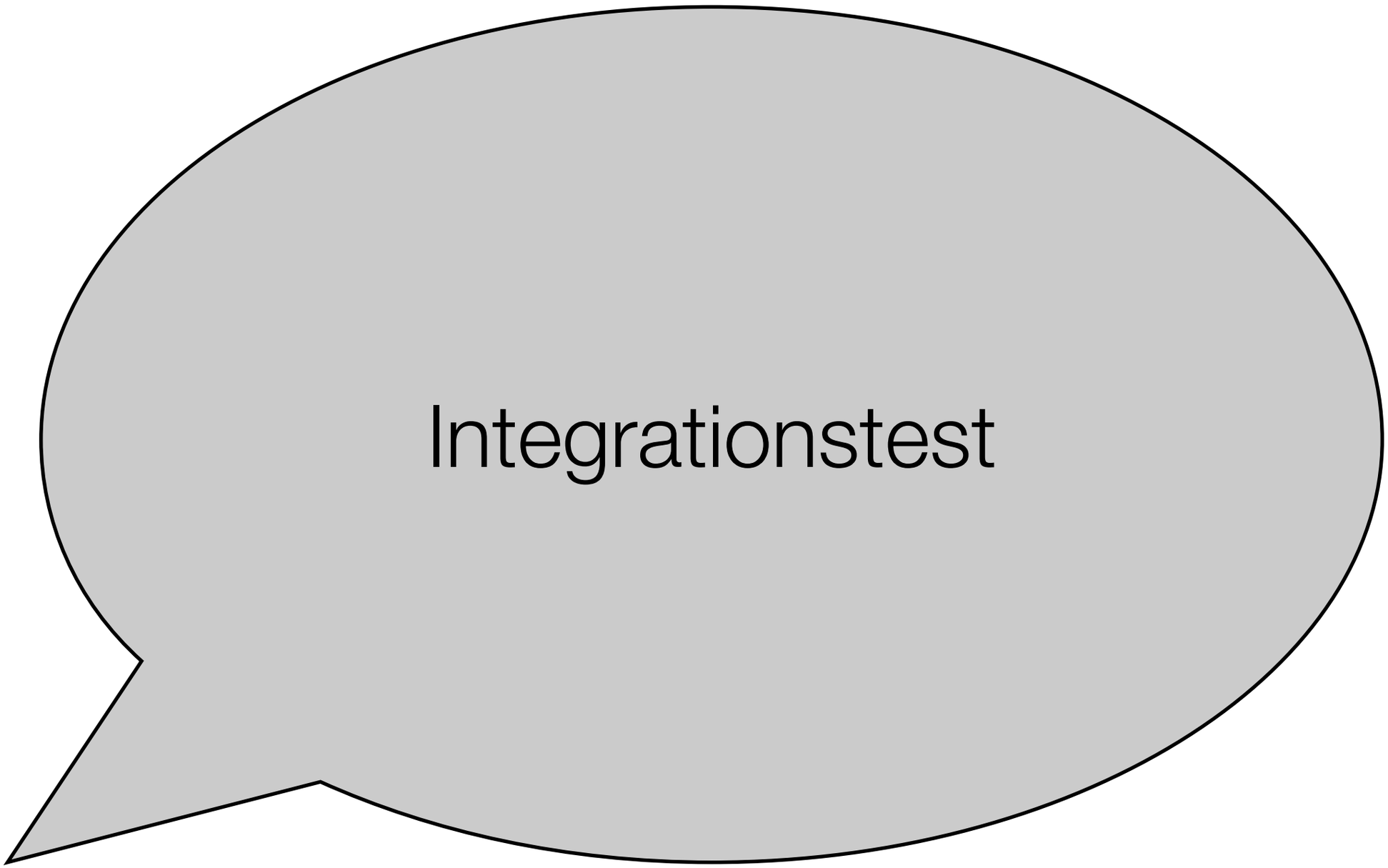
Voraussetzungen für Modultest

- hierarchische „Benutzt“-Relation
- keine Zyklen (→Zyklus durch eigenes Modul ersetzen)
- an der Spitze gibt es ein Modul, das von keinen anderen aufgerufen wird

Durchführen des Modultests

- analog zum Testen von Prozeduren (gegen Modulspezifikation)
- Treiber zum Aufruf des Moduls schreiben
- Stubs (Platzhalter) zum Simulieren der unterliegenden Module schreiben
  - Stub meldet nur Aufruf
  - Stub liefert konstantes Ergebnis
  - Stub ist auf Teilmenge des realen Moduls definiert




Integrationstest

Ansatz

- Programm in bestimmter Reihenfolge aus Modulen zusammenbauen
- das jeweils neu hinzugekommene Modul intensiv testen:
  - Schnittstellen zum vorhandenen Teilsystem
  - vollständiger Test des *hinzugekommenen* Moduls nach einem Überdeckungsmaß

→ Aus der Reihenfolge des Zusammenbauens ergeben sich verschiedene Teststrategien

„Big bang“-Methode

1. Teste alle Module einzeln für sich
2. Teste dann das komplette Programm

→ nicht inkrementeller Ansatz

Vorteile:

- Phase 1 ist gut parallelisierbar
- jedes Modul wird einzeln getestet

„Big Bang“-Methode

Nachteile:

- Maximale Anzahl von Treibern/Stubs benötigt (für jedes Modul)
- Schnittstellenfehler werden erst spät entdeckt
- Lokalisierung von Defekten im Gesamtsystem schwierig

→ Übergang zum inkrementellen Testen

Inkrementelles Testen (allgemein)

1. Teste zu Beginn ein beliebiges Modul
2. Füge ein Modul M hinzu mit folgenden Eigenschaften:
   a) M benutzt keine anderen Module, oder
   b) wenn M andere Module benutzt dann mindestens eines aus dem bestehenden Teilsystem
   c) wenn M von anderen Modulen benutzt dann mindestens von einem aus dem bestehenden Teilsystem
3. Wiederhole Schritt 2) bis Teilsystem = Gesamtsystem

Inkrementelles Testen (Spezialfälle)

- absteigender Test (top-down)
  - Beginne mit dem Modul an der Spitze der Hierarchie
  - Füge jeweils ein Modul hinzu, das von den vorhandenen benutzt wird
- aufsteigender Test (bottom-up)
  - Beginne mit einem Modul am „Boden“ der Hierarchie
  - Füge jeweils ein Modul hinzu, das nur Module benutzt, die zum bereits vorhandenen Teilsystem gehören

Vor- und Nachteile des iterativen Testens allgemein

+ top-down braucht keine Treiber

+ bottom-up braucht keine Platzhalter

+ Schnittstellenfehler werden beim Dazubinden von Modulen gefunden

+ Fehlerlokalisation wird einfacher (→ im neu hinzugekommenen Modul?)

+/- zuerst gewählte Module werden gründlicher getestet

- höherer Testaufwand, wächst mit größer werdendem Teilsystem

- weniger gut parallelisierbar

Entscheidungshilfe absteigender Test

- Probleme in oberen Modulen werden frühzeitig aufgedeckt

  → hilfreich zum Finden von Entwurfsproblemen

- Bei früher Einbindung der I/O-Module ist Darstellung der Testdaten einfach

  →sofern I/O „oben“ in der Modulhierarchie angesiedelt sind

- Es gibt jederzeit ein vorführbares „Programmskelett“

  → Motivation, verzahntes absteigendes Kodieren + Testen

  → Prototyping wird unterstützt

Nachteile absteigender Test

- Komplizierte Stubs
- Erzeugen von Testfällen für untere Module schwierig (lange Wege)
- Portabilität unterer Module schwierig zu testen
- Paralleles Entwerfen, Kodieren und Testen vermeiden (erst komplett entwerfen!)
  - Bei Implementierung der unteren Ebenen werden Änderungen weiter oben notwendig (iteratives Programmieren)
  - aber nicht oben erneut getestet

  →Tendenz schon implementierte obere Ebene so zu lassen und Entwurfsfehler durch Herumdoktern auf der unteren Ebene zu kaschieren
- Einbinden von Standardmodulen/Bibiliotheken nicht herausschiebbar

Entscheidungshilfe aufsteigender Test

- Entwurfs- und Kodierungsmängel in „unteren“ Modulen werden früh entdeckt
- Testfälle sind leicht zu erzeugen
  (kurze Wege - Treiber sitzt direkt auf dem zu testenden Modul)
- vollständiger Test der Module möglich
  (nicht nur der aktuell genutzte Funktionsumfang)
- Ergebnisse sind deshalb auch leichter zu überprüfen
- zu Beginn sind Tests teilweise parallelisierbar
- verhindert gleichzeitiges Entwerfen, Implementieren und Testen

Nachteile aufsteigender Test

- kein Programmskelett / vorführbarer Prototyp während der Entwicklung
- Defekte in oberen Modulen werden erst spät entdeckt

  →speziell für Benutzeroberfläche und -dokumentation ungünstig

- Typischerweise: aufsteigender Test der beste Kompromiß

Unterstützung des Integrationstests im Entwurf

1. Module mit wenigen Kopplungen an andere Module

   → Anzahl der Schnittstellen (-kombinationen) klein halten

   → (mehr Effizienz durch weniger Parameterübergaben)

2. Module mit hohem inneren Zusammenhang

   → weniger Parameter pro Schnittstelle (einfacher zu verstehen/testen)

3. Keine Verwendung von globalen Variablen

4. Nur Zugriff auf Module der nachfolgenden Schicht;
   keine indirekte Rekursion (Aufruf A-B-A-C)

   → Aufrufreihenfolgen müssen überschaubar bleiben

Spezielle Fehlerarten beim Integrationstest

Motivation

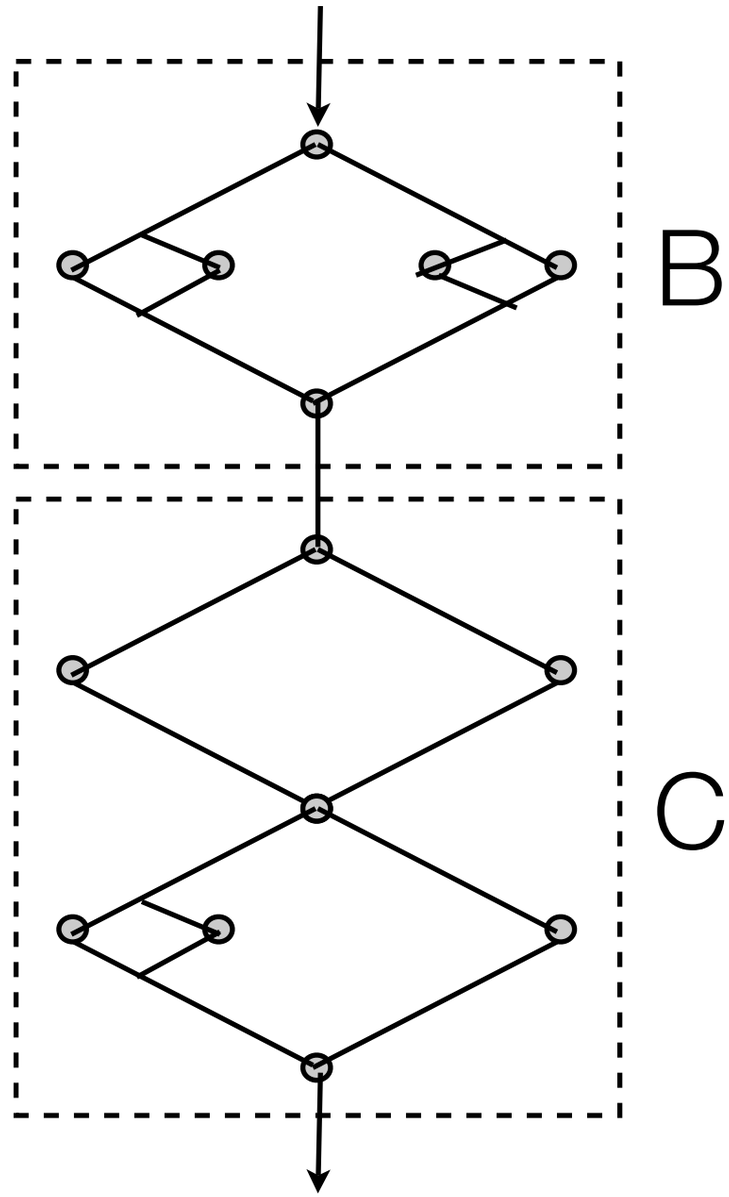
Modultest: gleiches Vorgehen wie bei Prozeduren anwenden

Integration von mehreren Modulen

- Aufwand für Pfadkombinationen etc. explodiert
- bei der Integration treten spezielle Fehlerklassen auf, die man bei isolierter Betrachtungsweise nicht hat

Aufwand bei der Verkettung von Operationen

- Modul A rufe B und C nacheinander auf
- 4 Wege innerhalb von B
- 6 Wege innerhalb von C



→ 4*6 = 24 Wege für alle Kombinationen

→ Allgemein: Verknüpfung von
m Operationen mit je n Wegen

= n^m Kombinationen (zu viele!)

→ Pfade in Modulen getrennt testen

im Beispiel: 1 + 4 + 6 = 11 Möglichkeiten

Gröbere Sichtweise beim Integrationstest

Abweichung des realisierten Teilsystems **vom Entwurf / Spezifikation**, z.B.:

- Kontroll- und Datenfluß
  - Reihenfolgebedingungen
- Inhalt und Form der Schnittstellen
- falsche Annahmen von Modulen über Funktionalität anderer Module
  - ist z.B. ein Sortierverfahren bei gleichen Schlüsseln reihenfolgeerhaltend?

Schnittstellenfehler

- syntaktische Fehler
  - falsche Anzahl / Typen von Parametern
- semantische Fehler
  - vertauschte Parameter kompatiblen Typs
- Aufruf mit undefinierten Eingabeparametern
- Spezialfall: Integrationszeit-Bereichsfehler

Integrationszeit-Bereichsfehler

```
read(i)
c := i;          (richtig wäre c := i+1)
P(c,a);
write(a);



P(mc,ma):
  if(mc<4) then ma := 1;
           else ma := 2;
```

i	FALSCH	richtig
...	1	1
2	1	1
3	1	2
4	2	2
5	2	2
...	2	2

- Defekt wird durch Betrachten der Einzelmodule nicht gefunden
- Fehleraufdeckende Teilbereich ist zudem sehr klein

Anpassung der Testmethoden für den Integrationstest

Ansatz

- Anpassung der vorhandenen statischen und dynamischen Methoden

Statische Methoden (1)

Syntaxprüfung der Schnittstellen

- Compiler / Linker

Vergleich realisierte Modulkopplungen ↔ Entwurf

- gewollte Abweichungen → Probleme im Entwurf
- ungewollte Abweichungen → Implementierungsfehler

→ Methoden der statischen Analyse

Statische Methoden (2)

- Aufdecken versteckter Abhängigkeiten
  - Modul A exportiert globale Variable an Module B und C
  - → ggf. unspezifizierte Abhängigkeit zwischen B und C

Statische Methoden (3)

- Intermodulare Datenflußanalyse - Beispiel:

```
...
int rueckgabe;
berechne(&rueckgabe);      /* Ergebnis in rueckgabe zurück */
rueckgabe=0;               /* dd-Anomalie */
```

- Markieren von Aufrufparametern als Rückgabewerte

  → Module können einzeln überprüft werden

- Allgemeiner Fall:

  - alle letzten Aktionen des Aufrufers / ersten Aktionen des Aufgerufenen bestimmen (beim Verlassen des Aufrufers umgekehrt)

  → übliche Verfahren für Datenflußanomalien einsetzen

  → vermeiden: globale Variablen, Durchreichen von Werten

Dynamische Methoden - Übersicht

- ablaufbezogener Test
  - kontrollfluß/datenflußbasiert
- wertbezogener Test
- funktionsbezogener Test

Ablaufbezogener Test

Kontrollflußkriterien - basierend auf dem Modulgraphen

1. alle Module
   - jedes Modul muß mindestens einmal aufgerufen werden

2. alle Relationen
   - durch jeden potentiellen Aufrufer mindestens einmal aufgerufen werden
     (= alle Kanten im Modulgraphen)

3. alle Relationen mehrfach
   - durch jeden potentiellen Aufrufer über alle möglichen Funktionen aufgerufen

4. alle Importe mehrfach
   - alle Aufrufstellen einer Modulfunktion testen

5. alle Aufrufreihenfolgen
   - jede mögliche Reihenfolge von Aufrufen über Modulgrenzen hinweg

Ablaufbezogener Test

Datenflußkriterien - Orientierung an Schnittstellen-Aufrufen

1. Statisch festgestellte Datenflußanomalien ausführen

2. Aufrufe an der Schnittstelle von Modulen testen:

   - für Schnittstellenvariablen erste/letzte Aktion vor/nach dem Aufruf bezüglich *define* / *reference* ermitteln

   - alle/einige Wege ausführen:
     - letztes *define* Aufrufer → erstes *reference* im Zielmodul
     - letztes *define* Zielmodul → erstes *reference* im Aufrufer

Wertbezogener Test

- Werte testen, die mit hoher Wahrscheinlichkeit Defekte aufdecken
- insbesondere solche, die beim Modultest mit Platzhaltern nicht testbar waren

a) Grenzwerte

b) Extremwerte (0, 1, ±MAXINT, ...)

c) Fehlerfälle und unzulässige Parameterkombinationen

Funktionsbezogener Test

Zusammenspiel der von den Modulen realisierten Teilfunktionen testen.

Ansatz: Abweichungen von der Spezifikation finden

- mangelnde Funktionalität

  - Modul liefert erwartete Teilfunktion nicht

- zuviel Funktionalität

  - Teilfunktion ist nicht spezifiziert / erwartet

- falsche Funktionalität

  - Testdaten aus ablauf-/wertbezogenen Tests nehmen
  - gegen Spezifikation vergleichen, ggf. zusätzlich Inspektion vornehmen

Testdatenerzeugung

Vorgabe: Modul M ruft importierte Prozedur P auf

1. Bestimme Testdaten aus dem Modultest von M, die P ausführen

2. Bestimme die resultierenden Parameterwerte für P

3. Wenn diese Werte einen geforderten Pfad / Kriterium für P erfüllen, nimm das Testdatum in die Testmenge auf

4. Erfüllt die Testdatenmenge jetzt noch nicht das festgelegte Kriterium, erzeuge die fehlenden Testdaten durch symbolisches Rückrechnen der fehlenden Wege / Berechnungen.

Ende der heutigen Vorlesung

Danke fürs Zuhören!

Bis nächste Woche :-)