

Software- Qualitätsmanagement

**Vorlesung im Modul 10-202-2319
Software-Management**

Sommersemester 2010

Prof. Dr. Hans-Gert Gräbe

<http://bis.informatik.uni-leipzig.de/HansGertGraebe>

Einführung

Quantitative Aussagen über die Produktqualität einer Systemkomponente können mit Hilfe von **Metriken** ermittelt werden.

- Mit solchen Metriken sind heute nur einfache Aussagen über Eigenschaften einer Komponente möglich.
- Eine Metrik bewertet ein Software-System immer nur unter einem sehr speziellen Blickwinkel.
- Aussagekräftiger Gesamteindruck von einer Systemkomponente nur durch Auswertung einer Gruppe von Metriken, oft auch nur im Vergleich zu Parametern anderer, bereits im Einsatz befindlicher Komponenten.
- Metriken können nicht nur für bereits implementierte Komponenten, sondern auch schon entwicklungsbegleitend eingesetzt werden.

Metriken zum Erfassen der prozeduralen Komplexität

Ziel: Bewertung eines Produkts (Entwurfsdokuments, Grob/Fein-Entwurf, Code, Designdokumentation usw.) mittels Metriken

Schwerpunkt: Zuverlässigkeit, Änderbarkeit

- **Umfangsmetriken**

- sind die ältesten Metriken
- stellen ab auf die textuelle Komplexität
- verwenden einfach verfügbare Informationen (Anzahl an Programmzeilen, Dateigröße, Zahl der Funktionen, ...)
- Vertreter: LOC, Halstead-Metrik, Function Points (zur Erfassung des Umfangs verbaler Anforderungen)

- **Logische Strukturmetriken (Kontrollfluss-Metriken)**
 - Analyse des Kontrollfluss-Graphen
 - Wird samt seiner Begleitobjekte (Symboltabelle) sowieso vom Compiler ausgewertet
 - Vertreter: McCabe-Metrik
- **Datenstrukturmetriken**
 - messen die Anzahl an Variablen, deren Gültigkeit und Lebensdauer sowie die Referenzierung der Variablen
- **Stilmetriken**
 - messen ob die Programme richtig eingerückt wurden und ob die Namenskonventionen eingehalten wurden
- **Interne Bindungsmetriken**
 - messen die syntaktische Bindung durch Prüfen des Codes jeder Komponente

Beispiele

Einsatzgebiet	Kriterium	Metrik
Komponenten-analyse	Umfang	lines of code
	innere Struktur	Kontrollfluss-komplexität
	Schnittstelle	# Methoden pro Klasse Schnittstellenbreite

7. Analysierende Verfahren

2. Metriken für Komponenten

Einsatzgebiet	Kriterium	Metrik
Systemanalyse	Umfang	lines of code
	Kopplung	# Aufrufe in/aus Komponenten
	OO-Strukturierung	OO-Metriken
Prozessanalyse	Aufwandsoptimierung	Zeiterfassung
	Dokumentenqualität	entdeckte Fehler pro Seite
	Prüfprozessqualität	# vorab gefundener Fehler / # in der Sitzung gefundener Fehler

Umfangsmetriken - Die Halstead-Metrik

- Misst die textuelle Komplexität eines Programms, indem die Zahl der verwendeten Funktionen und der verwendeten Variablen ins Verhältnis gesetzt werden.
 - Es wird jeweils die Gesamtzahl (Programmtext) und die Zahl verschiedener Objekte (Symboltabelle) bestimmt.
 - η_1, N_1 = Zahl der (verschiedenen) Funktionen, Operatoren, Symbole oder Schlüsselwörter (z. B.: +, -, *, /, **while**, **if**, ...)
 - η_2, N_2 = Zahl der (verschiedenen) Variablen, Operanden ...
- **Interpretation:**
 - $\eta = \eta_1 + \eta_2$: Größe des Vokabulars
 - $N = N_1 + N_2$: Länge der Implementierung

- **Vorteile:**
 - einfach zu ermitteln,
 - bei jeder Programmiersprache verwendbar und
 - gute Eignung der Metriken für die zu messenden Größen
- **Nachteile:**
 - nur der Implementierungsaspekt betrachtet und
 - Mehrdeutigkeiten im Messansatz, z. B. bei den Klassifikationsregeln für Operatoren und Operanden
- **abgeleitete Größen:**
 - $D = \eta_1 / 2 \cdot N_2 / \eta_2$,
Parameter für die Schwierigkeit, den Code zu verstehen
 - Interpretation:
 N_2 / η_2 = durchschnittliches Vorkommen jeder Variablen,
 η_1 = Anzahl der verwendeten Funktionen


```
int ZaehleVokale(String s) {  
    int VokalAnzahl; char Zchn; int i;  
    for(i=0; i < s.length(); i++) {  
        Zchn=s[i];  
        if ((Zchn == 'A') || (Zchn == 'E') || (Zchn == 'I') ||  
            (Zchn == 'O') || (Zchn == 'U')) VokalAnzahl++;  
    }  
    return VokalAnzahl;  
}
```

ZaehleVokale	1	int	3	()	9	++	2
VokalAnzahl	3	String	1	{}	2	[]	1
Zchn	7	char	1	;	8	==	5
s	3	for	1	=	2		4
i	5	if	1	<	1	.	1
Konstanten (6)	6	return	1	length	1		

$$\eta_1=17, N_1=44, \eta_2=11, N_2=25, D=19.32$$

Kontrollfluss-Metriken - Die McCabe-Metrik

- Misst die strukturelle Komplexität eines Programms, indem eine Grapheninvariante, die **zyklomatische Zahl** $V(G)$ des Kontrollflussgraphen, bestimmt wird.
- $V(G) = e - n + 2p$ mit
 - e = Anzahl der Kanten des Graphen
 - n = Anzahl der Knoten
 - p = Anzahl der Zusammenhangskomponenten
- Kontrollflussgraph wird für jede Prozedur aufgestellt ($p=1$).
- Für solchen Graphen gilt

$$V(G) = b + 1$$

mit b = Anzahl der Bedingungen.

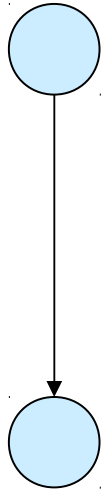
- Zyklomatische Zahl ist additiv auf Komponenten.
- Lineare Teilstücke können zusammengezogen werden.

- **Vorteile:**
 - einfach zu berechnen,
 - grobes Maß für die Kontrollflusskomplexität: je größer, desto weiter weicht der Kontrollfluss vom linearen ($V(G)=1$) ab.
- **Nachteile:**
 - unterschiedliche Programmmerkmale werden stark vereinfacht
 - Quellprogramm als zentrales Messobjekt überbetont
 - Es wird nur das Programmgerüst, nicht aber die Komplexität einzelner und verschachtelter Anweisungen berücksichtigt

7. Analysierende Verfahren

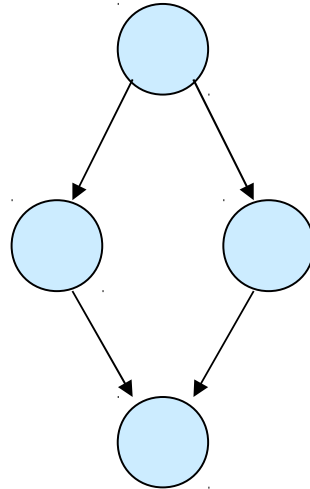
2.2 McCabe-Metrik - Beispiel

Sequenz



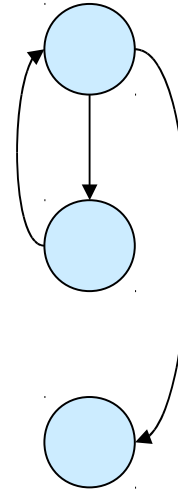
$$V(G) = 1 - 2 + 2 \\ = 1$$

Auswahl

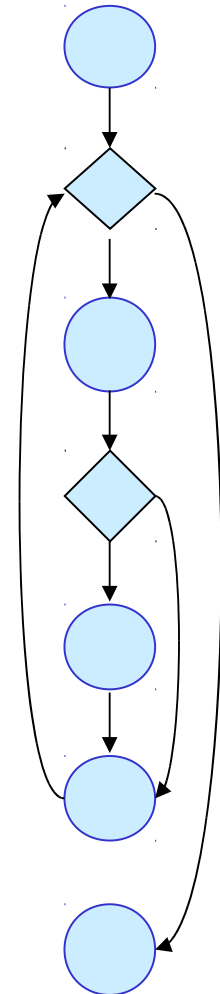


$$V(G) = 4 - 4 + 2 \\ = 2$$

Abweisende Schleife



$$V(G) = 3 - 3 + 2 \\ = 2$$



Das Programm ZaehleVokale hat die zyklomatische Zahl $V(G) = 8 - 7 + 2 = 3$
Es enthält zwei Bedingungen.

Metriken für objektorientierte Entwicklung

- Metriken der klassischen Software-Entwicklung sind in unveränderter Form für OO-Projekte nur bedingt aussagefähig.
- Umfangsmetriken: Wie mit geerbtem Code umgehen?
 - Durch Vererbung und Polymorphismus sinkt die Zeilenzahl signifikant
- McCabe-Metrik: Kontrollflusskomplexität bei OO meist sehr gering
- Frage nach Maßen für die OO-spezifischen Effekte
 - zusätzliche Maße waren erforderlich
 - Breite und Höhe der Vererbungshierarchie
 - Anzahl der Klassen, die eine spezielle Operation erben
 - Anteil wieder verwendeter Komponenten
 - Anzahl der Objekt- und Klassenattribute
 - Anzahl der Objekt- und Klassenoperationen

Typische Metriken (Beispiele)

- Objekt- und Klassenattribute einer Klasse:
 - Anzahl: $|OV|$, $|CV|$
 - gewichtete Anzahl: $\sum T(v)$
 - $T(v)$ = Gewicht des Typs des Attributs v
 - gewichtet nach Zahl der Vorkommen, nach Zahl der Vorkommen in verschiedenen Methoden der Klasse etc.
- Objekt- und Klassenmethoden einer Klasse:
 - $|OM|$, $|CM|$, evtl. wieder gewichtet nach Komplexität
 - Parameterkomplexität einer Methode
 - Zahl und Gewicht der Typen der Aufrufparameter
 - McCabe-Metrik
- durchschnittliche Komplexität von Klassen in einem Paket
 - Durchschnittswerte der einzelnen Klassenparameter
 - Kreuzreferenzparameter (Bindungsanalyse im Paket)

Folgende Metriken werden in der Literatur als signifikant genannt

- DIT (*Depth of Inheritance Tree*):
 - Tiefe des Vererbungsbaumes (Zahl der Vorfahren einer Klasse)
 - je höher der Wert von DIT, desto höher die Fehlerwahrscheinlichkeit (Hoher Nachnutzungsgrad, Nichtbeachtung verdeckter Annahmen)
- NOC (*Number of Children of a Class*):
 - Anzahl der direkten Nachfolger einer Klasse
 - je höher der Wert von NOC, desto geringer die Fehlerwahrscheinlichkeit (sehr präzise Abstraktion, Nichtvorhandensein verdeckter Annahmen)
- RFC (*Response For a Class*):
 - Anzahl der Funktionen, die direkt durch die Methoden einer Klasse aufgerufen werden.
 - je höher der Wert von RFC, desto größer die Fehlerwahrscheinlichkeit (Hoher Delegationsgrad, Nichtbeachtung verdeckter Annahmen)

- WMC (*Weighted Methods per Class*):
 - Anzahl aller neu definierten oder überschriebenen Methoden, die in jeder Klasse definiert sind.
 - Geerbte Methoden werden nicht gezählt.
 - je größer der Wert von WMC, desto höher die Fehlerwahrscheinlichkeit
- CBO (*Coupling Between Object Classes*):
 - eine Klasse ist mit einer anderen Klasse gekoppelt, wenn sie deren Methoden und/oder Attribute benutzt.
 - CBO ist die Anzahl der Klassen, mit der eine Klasse gekoppelt ist.
 - je größer der Wert von CBO, desto größer die Fehlerwahrscheinlichkeit (höherer Verschränkungsgrad)

Vorteile:

- erste Ansätze zur Verbesserung objektorientierter Komponenten
- erste empirische Untersuchungen zeigen die Eignung einiger Metriken als Qualitätsindikatoren

Nachteile:

- die Ziele der Metriken sind nur implizit erkennbar
 - Zielrelevanz der Metriken noch ungenügend untersucht
- keine Metriken für dynamische Aspekte
 - z. B.: Zustandsautomaten
- keine semantische Unterscheidung der Methoden
 - Standardoperationen (lesen, schreiben, erzeugen,...) sind weniger fehleranfällig als auszuprogrammierende "fachkonzeptspezifische" Methoden
- keine Berücksichtigung der Oberklassenqualität
 - ob eigene, geerbte oder fremde Methoden
 - ob Vererbung von Methoden aus Standardklassen oder eigenen
- keine Metriken, die eine "gute" Vererbungsstruktur prüfen

Eine **Anomalie** ist jede Abweichung bestimmter Eigenschaften eines Programms von der korrekten Ausprägung dieser Eigenschaften.

- Konstruktive Sprachkonzepte erlauben das Aufdecken von Anomalien durch statische Quelltextanalyse
 - Beispiel: Typprüfung durch den Compiler
 - Oft ist keine unmittelbare Fehlererkennung, jedoch eine Identifikation von Fehlerort und -symptomen möglich.

Datenflussanomalien-Analyse

- **Ziel:** Aufdecken von Datenflussanomalien durch Analyse von Programmpfaden auf sinnvolle Datenfluss-Sequenzen

Datenfluss-Eigenschaften von Variablen

- Auf eine Variable x kann entlang eines Programmpfades wie folgt zugegriffen werden:
 - x wird definiert (d),
 - x wird referenziert (r),
 - x wird undefiniert (u) (z.B. beim Verlassen einer Methode)
 - x wird „geleert“ (e), d.h. der Wert an einen anderen Ort übertragen
- Enthält die Sequenz Teile, die keinen Sinn ergeben, so liegt eine Datenfluss-Anomalie vor.
 - Beispiele:
 - **rdru** ⇒ die Sequenz beginnt mit einer Referenz, vor der Definiton, diese Anomalie ist vom Typ **ur**
 - **ddrdu** ⇒ diese Sequenz beginnt mit einer doppelten Definition (Anomalientyp **dd**) und endet mit **du**

Beispiel

```
/* swap (int a, int b) */  
int hilf; a=hilf; a=b; hilf=b;
```

Analyse:

a d : d d : r

b d : r r : r

hilf u : r d : e

Anomalietyp:

mehrfach nacheinander überschrieben

nie verändert

neu definiert vor e

```
/* swap (int a, int b) korrigierte Version */  
int hilf; hilf=b; b=a; a=hilf;
```

a d : r d : r

b d : r d : r

hilf u : d r : e

Leistung:

entdeckt Wertzuweisungen an falsche Variablen, Anweisungen an unkorrekter Stelle und fehlende Anweisungen

Vorteile:

- sichere Entdeckung bestimmter Fehlertypen,
- geringer Aufwand im Vergleich zu dynamischen Verfahren,
- direkte Fehlerlokalisierung und
- gute Ergänzung zu anderen Testverfahren

Nachteile:

- Leistungsfähigkeit auf schmalen Fehlerbereich begrenzt

Analysierende Verfahren - Zusammenfassung

- Quellcodeanalyse
 - Bindung und Kopplung als qualitatives Maß für „guten Code“
 - Bindung auf der Ebene einzelner Funktionen
 - Bindung auf der Ebene von Datenabstraktionen
 - Bindung auf der Ebene von Datenmodellen
 - Bindung in Vererbungshierarchien
- Komponentenanalyse
 - Umfangsmetriken (Beispiel: Halstead-Metrik)
 - Strukturmetriken (Beispiel: McCabe-Metrik)
 - Bindungsmetriken
 - welche Funktion wird wo und wie oft aufgerufen?
 - OO-Metriken
- Anomalienanalyse

Systemqualität

- Die Produktqualität eines SW-Produktes wird durch die Qualität seiner Komponenten (**Komponentenqualität**) und deren Beziehungen untereinander (**Systemqualität**) bestimmt.
- Systemqualität ergibt sich wieder aus dem Wechselspiel von konstruktiven und analytischen Maßnahmen
 - durch konstruktive Maßnahmen werden Defekte von Anfang an vermieden
 - durch analytische Maßnahmen werden Defekte aufgedeckt und beseitigt
- Vor der Sicherung der Systemqualität steht die Sicherung der Qualität der beteiligten Komponenten

Überprüfung der Systemqualität erfolgt, zeitlich nacheinander, in drei Teststufen:

1. Integrationstest

- Test der Bindungen zwischen den Komponenten
- Vergleichbar mit Strukturtest in Komponenten
- System wird als White Box betrachtet

2. Systemtest

- Test des Systems als Ganzes gegen die Spezifikation
- Vergleichbar mit Funktionstest in Komponenten
- System wird als Black Box betrachtet

3. Abnahmetest

- Systemtest in der Anwendungsumgebung und unter Beteiligung des Auftraggebers
- endet mit formalen Abnahmeverfahren

Ziel:

fehlerfreies Zusammenwirken der System-Komponenten
überprüfen.

Voraussetzung:

Jede Systemkomponente muss vorher für sich allein getestet
worden sein.

Generelle Vorgehensweise:

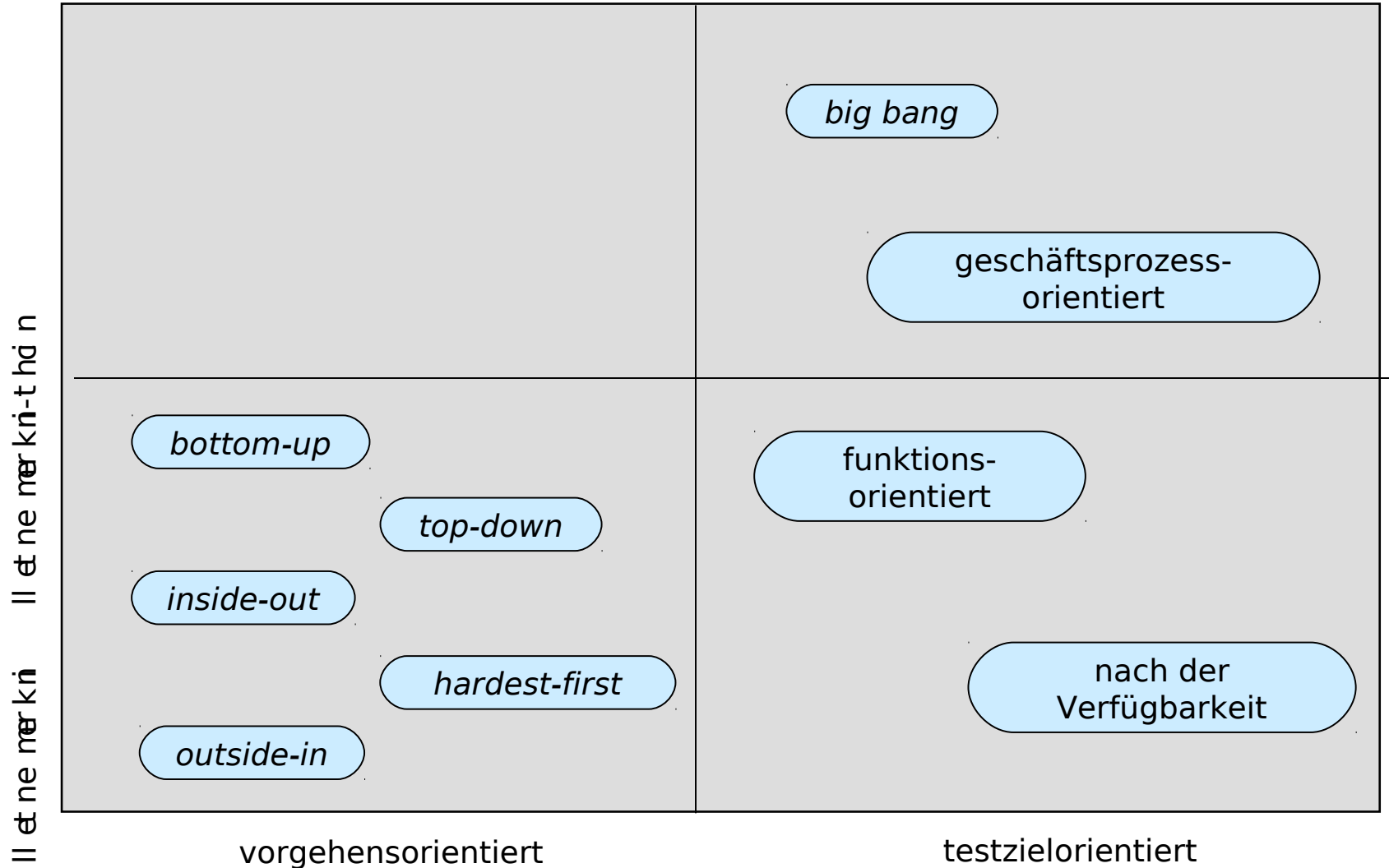
- System-Komponenten werden schrittweise zusammengesetzt mit regelmäßiger Überprüfung auf Fehler nach jeder neuen Komponente.
- Vorgehensweise (**Integrationsstrategie**) hängt vom verwendeten Entwicklungsmodell ab.

Integrationsstrategien

- zeitliche Reihenfolge, in welcher fertiggestellte und überprüfte Systemkomponenten zu einem Gesamtsystem integriert werden.
- Unterscheidung zwischen inkrementellen und nicht inkrementellen Strategien
- Unterscheidung zwischen vorgehensorientierten und testzielorientierten Strategien

Integrationstest

- Testaktivität, welche begleitend zur Integration das korrekte Zusammenwirken der einzelnen Systemkomponenten überprüft.
 - erfolgt parallel zur Systemintegration
 - verwendet weitgehend modifizierte Überprüfungsverfahren, die auch für den Komponententest eingesetzt werden
 - spezielle Testschnittstellen: Platzhalter und Testtreiber



Verschiedene Integrationsansätze

nicht-inkrementelle Integrationsstrategien

- sehr viele oder sogar alle System-Komponenten (evtl. aus einem Teilbereich) werden gleichzeitig integriert
- Vorteil: keine Platzhalter oder Testtreiber nötig
- Nachteile:
 - alle System-Komponenten müssen zur Integration zur Verfügung stehen
 - Fehler sind schwer zu lokalisieren
 - Testüberdeckung schwierig zu realisieren, da geeignete Testfälle schwer zu konstruieren sind
- Beispiel geschäftsprozess-orientierte Integration:
 - Integration derjenigen Komponenten, die zu einem Geschäftsprozess gehören
- Beispiel big bang:
 - unsystematisch alles auf einmal

Inkrementelle Integrationsstrategien

- die System-Komponenten werden einzeln oder in kleinen Gruppen integriert
- (Noch) nicht verfügbare Komponenten werden durch Testtreiber oder Platzhalter ersetzt
- Vorteile:
 - die Komponenten können integriert werden, sobald sie fertig sind
 - leicht konstruierbare Testfälle, Testüberdeckung sichergestellt
- Nachteil: u. U. sind viele Testtreiber oder Platzhalter nötig
- Beispiel testziel-orientierte Integrationsstrategien:
 - Testfälle werden anhand der Testziele erstellt, etwa Testziel „frühzeitige Integration fertiger Komponenten“
 - zur Überprüfung dieser Testfälle baut man dann die dafür benötigten System-Komponenten zusammen
- verschiedene Vorgehensweisen sind möglich
 - top-down, bottom-up, hardest-first, ...

Testtreiber und Platzhalter

Testtreiber (*driver*)

- Spezielle Testschnittstelle, um System-Komponenten zu testen, deren Dienste nicht direkt von der Benutzungsoberfläche aufgerufen werden können.
- **Beispiel:** Funktionalitätstest einer Datenbankanbindung. Testtreiber liefert die Eingaben und prüft die Antworten

Platzhalter (*dummies, stubs*):

- Vorläufiger Ersatz für eine andere Komponente, die für den Test einer Systemkomponente benötigt wird, im Moment des Tests aber noch nicht verfügbar ist.
- **Beispiel:** Fehlende Komponente soll Inventurliste ausdrucken. Platzhalter übernimmt das, aber ohne Formatierung.

Im Allgemeinen sind Testtreiber leichter zu realisieren als Platzhalter.

top-down

- Prüfung der System-Komponenten beginnend von der Wurzel der Baum- oder Schichtenhierarchie
- schrittweise Integration, fehlende System-Komponenten werden simuliert
- Vorteile:
 - Frühzeitiges Simulationsmodell führt aus Sicht des Benutzers bereits einen Teil der Funktionen des endgültigen Systems aus.
 - Änderungen, Alternativen, Verbesserungen frühzeitig sichtbar.
 - Gezielte Prüfung der Fehlerbehandlung bei fehlerhaften Rückgabewerten möglich, da Rückgabewerte aus Platzhaltern stammen.
 - Verzahnung von Entwurf und Implementierung ist möglich.
- Nachteile:
 - Platzhalter sind nötig (zusätzlicher Erstellungsaufwand).
 - Bei zunehmender Integrationstiefe steigt die Schwierigkeit, bestimmte Testsituationen zu erzeugen.
 - Zusammenwirken von zu prüfender Software, Systemsoftware und Hardware wird sehr spät untersucht.

bottom-up

- Zuerst werden die Basiskomponenten integriert, da diese keine Dienste anderer Komponenten benötigen.
 - Bei einer Baumhierarchie fängt man somit bei den Blättern an
- Vorteile:
 - keine Platzhalter nötig, leicht herstellbare Testbedingungen
 - Testergebnisse sind leichter zu interpretieren
 - bewusste Fehleingaben zur Prüfung der Fehlerbehandlung möglich
 - Zusammenwirken von Systemsoftware, Hardware und zu prüfender Software wird früh getestet
- Nachteile:
 - Testtreiber erforderlich
 - Fehler in der Produktdefinition werden erst spät gefunden, da lauffähiges Gesamtsystem erst am Ende verfügbar
 - gezielte Überprüfung der Fehlerbehandlung für Rückgabewerte ist kaum realisierbar, da die realen Komponenten benutzt werden

outside-in

- Kombination aus top-down und bottom-up, um die Vorteile beider zu vereinen und die Nachteile zu minimieren
- man beginnt gleichzeitig von oben sowie von unten und arbeitet zur Mitte hin

inside-out

- dieselbe Überlegung, aber man beginnt mit den System-Komponenten in der Mitte der Hierarchie und arbeitet nach oben und nach unten
- vereint eher die Nachteile von *top-down* und *bottom-up*, daher nur u. U. mit hardest-first einsetzen

hardest-first

- zuerst werden die kritischen, d.h. potenziell fehlerhaften und am schwierigsten zu implementierenden System-Komponenten implementiert und getestet
- damit wird diese Komponente besonders oft getestet