Qualitätssicherung und Testen

Dr. Friederike Nickl

friederike.nickl@sepis.de

Softwareentwicklung,
Produkt- und Informations-
Service GmbH

Berliner Straße 85,
80805 München

Gliederung

1. Qualitätsmanagement und Qualitätssicherung
2. Prinzipien der Software-Qualitätssicherung
3. Verfahren der analytischen Qualitätssicherung
   - 3.1 Statische Prüfverfahren: Inspektionen und Reviews
   - 3.2 Dynamische Prüfverfahren: Testen
     - 3.2.1 Black-Box Test
     - *3.2.2* White-Box-Test
4. Tests im SW-Entwicklungsprozess
   - 4.1 Unit-Test
   - 4.2 Integrationstest
   - 4.3 Systemtest
5. Prinzipien systematischen Testens

Qualitätsmanagement und Qualitätssicherung

Wiederholung: Qualitätsmerkmale von Software

- Korrektheit
- Zuverlässigkeit
- Robustheit
- Benutzerfreundlichkeit
- Wartbarkeit
- Effizienz (Performanz)
- Dokumentation
- Portabilität

Qualitätsmanagement

Alle Tätigkeiten, um die Qualität von Prozessen und Produkten sicherzustellen.

- Qualitätsplanung

  Festlegung von überprüfbaren Qualitätszielen und Planung von Maßnahmen zur Erreichung dieser Ziele (dokumentiert im QS-Plan)

- Qualitätssicherung

  Umsetzung der Maßnahmen des QS-Plans

Die Qualitätsziele können erreicht werden durch

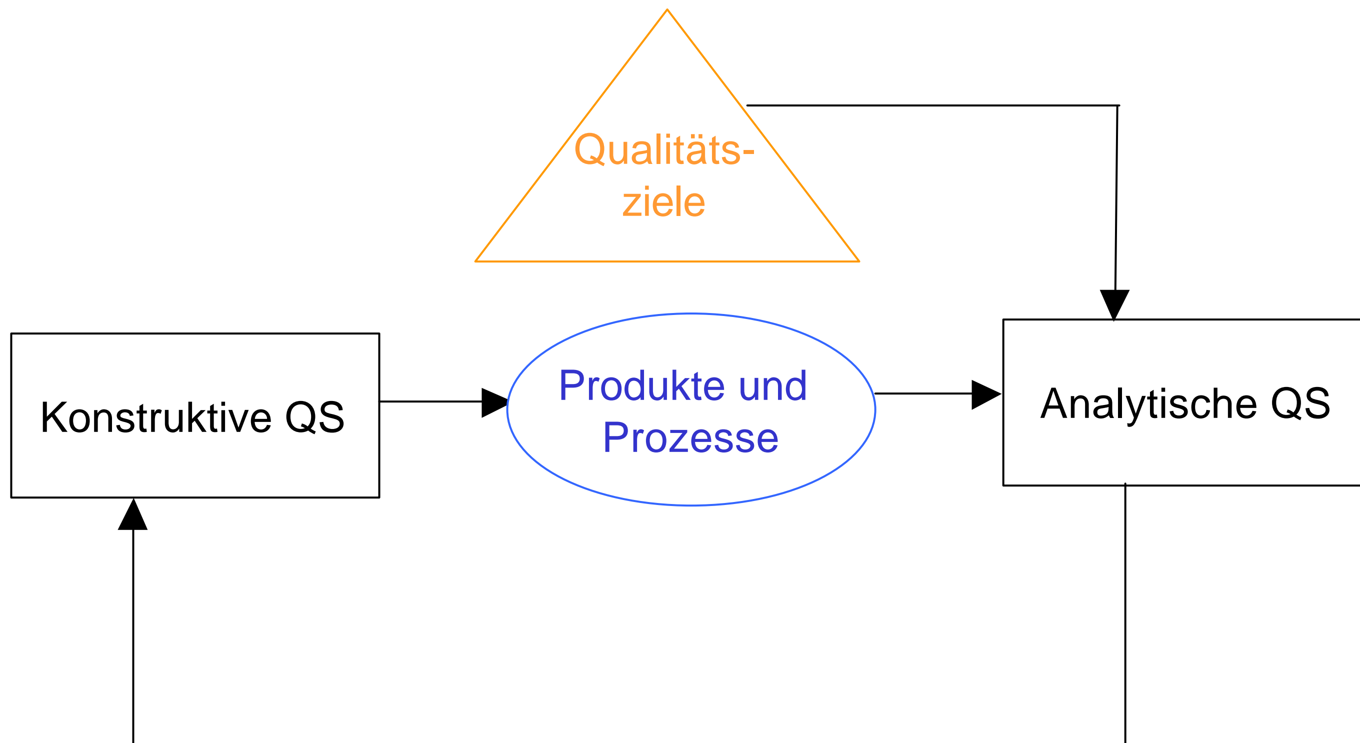
- **konstruktive QS-Maßnahmen**:

  Methoden, Sprachen, Werkzeuge, Richtlinien, Checklisten die dafür sorgen, dass das Produkt bzw. der Erstellungsprozess bestimmte Eigenschaften besitzt

- **analytische QS-Maßnahmen**:

  Prüfung und Messung des existierenden Qualitätsniveaus. Die Qualität wird gemessen, aber nicht verbessert.

Der **Qualitätszyklus**



2. Prinzipien der SW-Qualitätssicherung

Prinzip der Qualitätszielbestimmung

Festlegung von Qualitätszielen vor Beginn der Entwicklung

- das Team weiss, wohin es arbeitet
- der Kunde weiss, worauf er sich einläßt

Die Qualitätsziele werden im QS-Plan festgelegt

- die Kritikalität des Systems ist ein wichtiger Anhaltspunkt für die Festlegung der Ziele

Prinzip der quantitativen QS

- Ingenieurmäßige Qualitätssicherung ist undenkbar ohne die Quantifizierung von Soll- und Istwerten (Rombach, 93)
- Die Qualitätsziele müssen überprüfbar und messbar sein

Beispiel:

Kein überprüfbares Ziel: Das System muss performant sein

Überprüfbar: Die Verarbeitungszeit eines Vertrages durch das System muss unter 8 Sekunden liegen

Prinzip der maximal konstruktiven QS

Der Aufwand für die analytische QS soll durch eine möglichst gute konstruktive QS gering gehalten werden

- besser vorbeugen als heilen
- Fehler, die nicht gemacht werden können, brauchen auch nicht behoben zu werden

Beispiel

Beim Einsatz einer Programmiersprache mit statischer Typprüfung können Typfehler während der Laufzeit nicht auftreten

Prinzip der frühen Fehlerkennung und -behebung

- Fehler in den frühen Projektphasen sind die teuersten
  - Anforderungen des Kunden wurden nicht richtig verstanden
  - Inkonsistenzen im Anforderungsdokument
- Eine verzögerte Fehlerentdeckung führt zu einem exponentiellen Kostenanstieg.

⇒ Viel Aufmerksamkeit in die frühen Phasen der SW - Entwicklung investieren
  - Fehler durch konstruktive Maßnahmen verhindern
  - Fehler, die dennoch gemacht werden, durch sorgfältige Prüfungen der Dokumente in den frühen Phasen rechtzeitig erkennen

Prinzip der unabhängigen QS

- Ziel der analytischen Qualitätssicherung ist es, Fehler und Mängel aufzudecken
- Myers: „Testing is a destructive process, even a sadistic process“
- Entwickler können nicht gleichzeitig konstruktiv und destruktiv denken

⇒ die Entwickler eines Teilprodukts sollten nicht die analytische QS für dieses Teilprodukt vornehmen

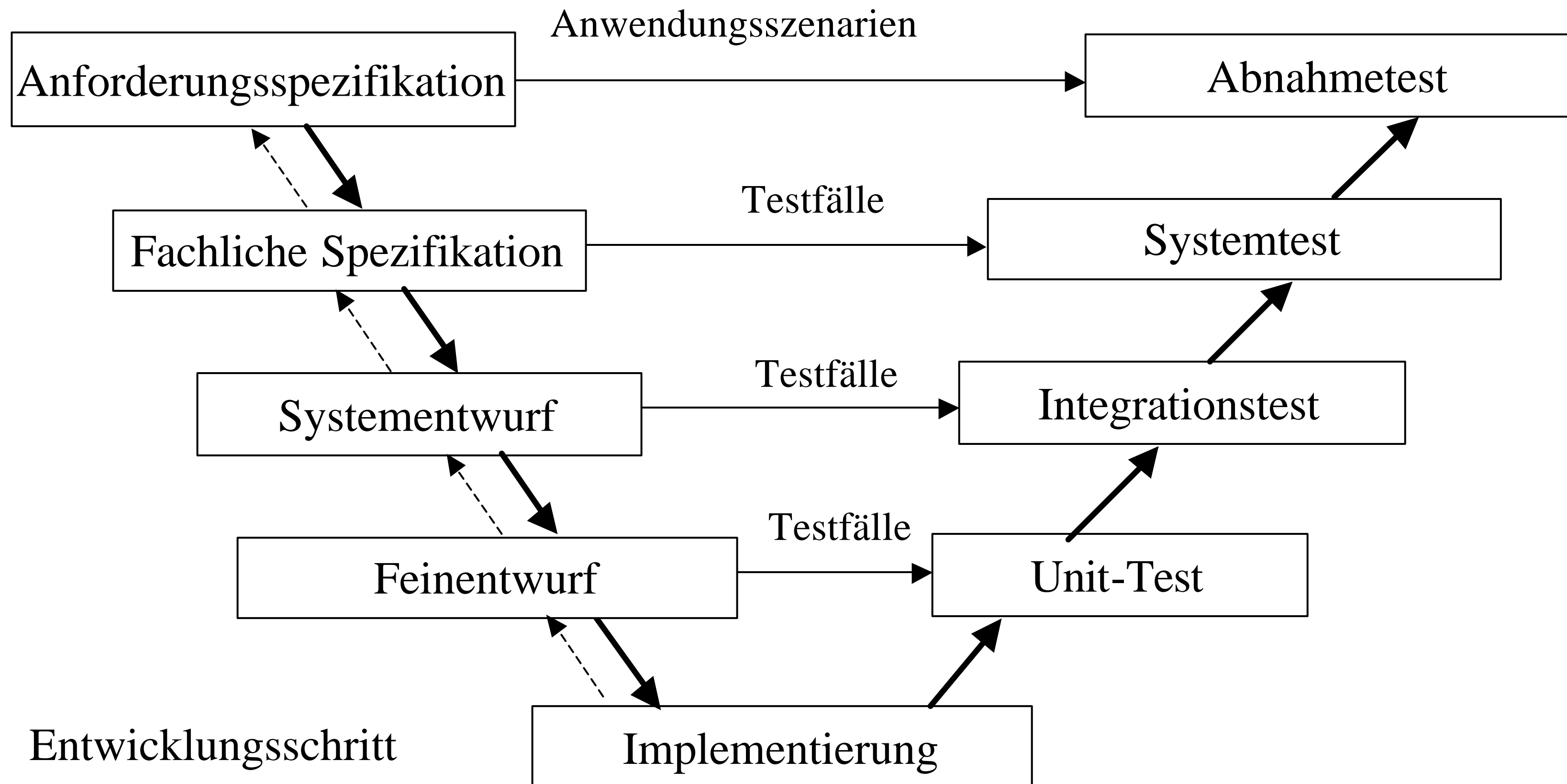
Prinzip der entwicklungsbegleitenden SW-QS

- Einbettung der QS in den organisatorischen Ablauf der SW-Entwicklung

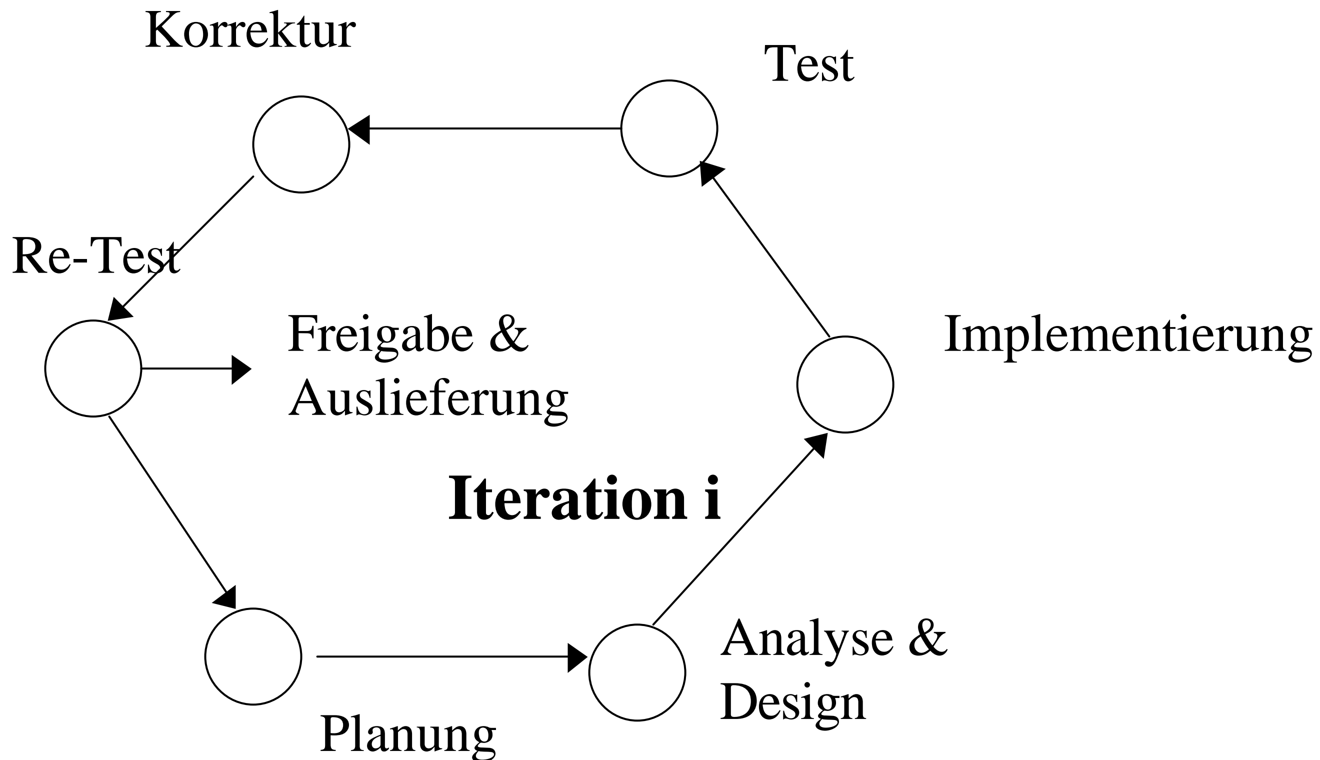
- Ein Teilprodukt steht der nächsten Phase erst dann zur Verfügung, wenn eine bestimmte Qualität erreicht ist:

- Beispiele
  - V-Modell
  - iterative SW-Entwicklung

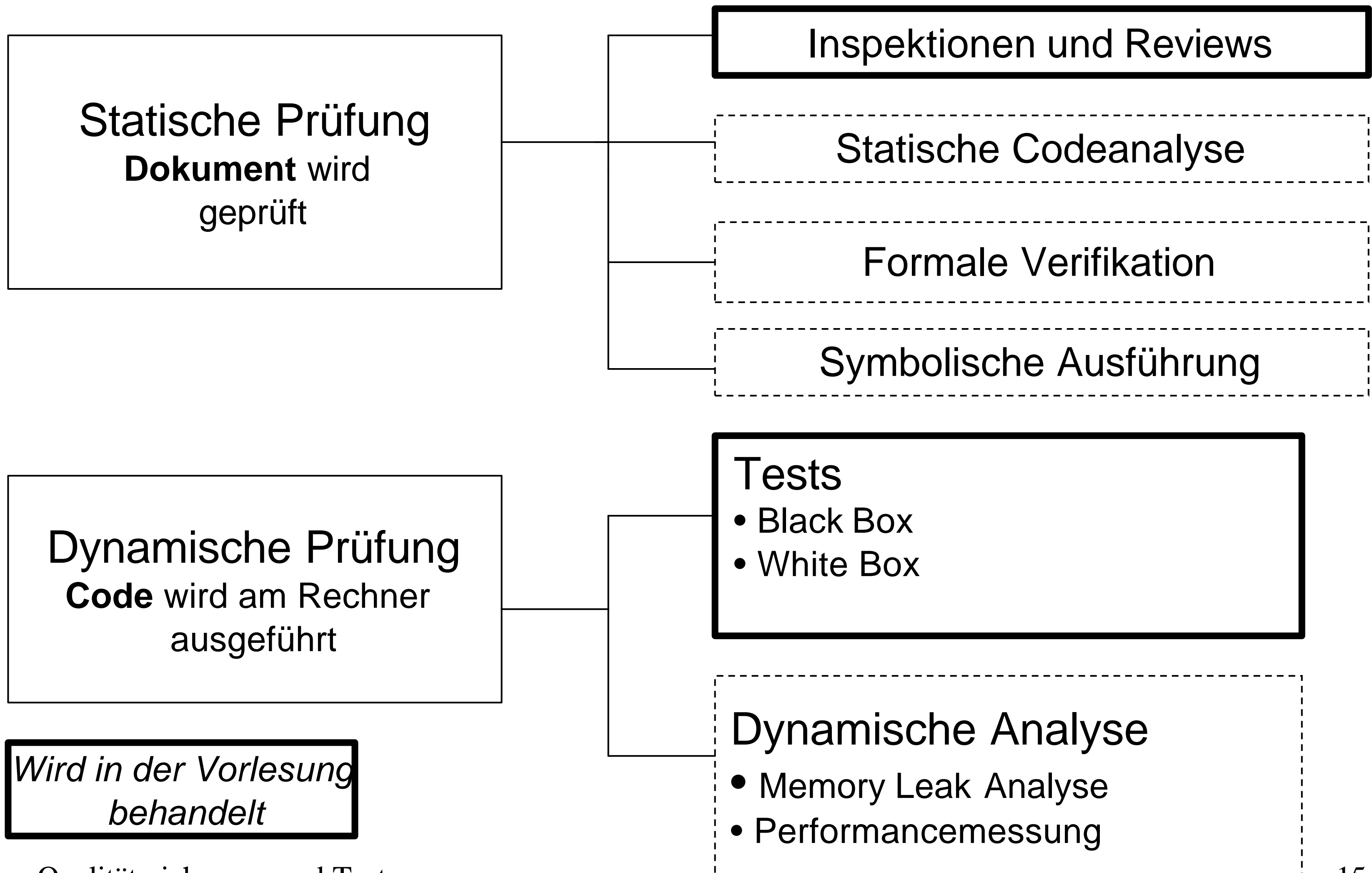
V-Modell: Integration der QS ins Wasserfallmodell



Einbettung der QS in die iterative SW-Entwicklung



3 Verfahren der analytischen Qualitätssicherung



3.1 Inspektionen und Reviews

Prüfmethode der Inspektion

- entwickelt von M.E. Fagan bei der IBM (1976)
- formalisierte Evaluationstechnik zur Überprüfung von
  - Softwareanforderungen
  - Entwurf
  - oder Code
- Überprüfung erfolgt durch eine Gruppe mit Moderator
  - der Moderator ist kein Vorgesetzter der Autoren!
- Ziele:
  - Aufdecken von Fehlern und Verletzungen von Standards im Dokument
  - Entscheidung über Freigabe

Ablauf einer Inspektion

- Auswahl eines Moderators
- Eingangsprüfung
- Planung
  - *Wer*: Inspektionsteam
  - *Wogegen*: Vorläufer-Dokumente, Checklisten, Erstellungsregeln
  - *Wann*: Termine
- individuelle Vorbereitung und Prüfung durch die Inspektoren
  - Arbeitsgeschwindigkeit wird empfohlen (z.B. 1 Seite/Std)
- Inspektionssitzung mit Ergebnisprotokoll
- Überarbeitung (Autoren)
- Nachprüfung und Entscheidung über Freigabe (Moderator)
- Zusammenstellung statistischer Daten über die Inspektion (Moderato

Inspektionen von Anforderungsspezifikationen

- Inspektoren sind Auftraggeber, Fachexperten, IT-Betrieb, Entwickler (Stakeholders)
- die Spezifikation ist zu prüfen
  - gegen die Wünsche und Anforderungen der Auftraggeber, Anwender und Fachbereiche
  - auf Konsistenz der Anforderungen
  - auf Durchführbarkeit in der gegebenen fachlich/organisatorischen und technischen Umgebung
- Kosten für die Inspektion zahlen sich aus, da Fehler in den Anforderungen teuer zu stehen kommen.

Inspektionen von Code

- das zu prüfende Dokument ist ein Stück zusammenhängenden Codes
- Inspektoren sind Realisierer im Team
  - Streuung von Know-How über den Code
- Prüfung des Codes gegen die Spezifikation und gegen Programmierrichtlinien
- Es lassen sich Fehler entdecken, die in Tests nur schwer zu provozieren sind

Inspektionen: Empirische Ergebnisse

- Es gibt zahlreiche Veröffentlichungen über empirische Ergebnisse von Software-Inspektionen
- Folgende Faustregel wird darin bestätigt
  - 50 bis 75% aller Entwurfsfehler können durch Inspektionen gefunden werden
  - Code-Inspektionen sind ein sehr kosteneffektiver Weg, um Defekte auszugleichen
- Beispiel: Studie bei der NASA

  Aufdecken von Fehlern pro Std. Aufwand
  - durch Code-Reading: 3,3 Fehler
  - durch Test: 1,8 Fehler

Reviews

- Ein Review ist ein mehr oder weniger formalisierter Prozeß zur Überprüfung von schriftlichen Dokumenten durch Gutachter.
- Reviews haben im wesentlichen den gleichen Ablauf wie Inspektionen, sind aber weniger formalisiert.

3.2 Dynamische Prüfverfahren: Testen

- **Testen** ist der Prozess, ein Programm auf systematische Art und Weise auszuführen, um Fehler zu finden
- Ein **Fehler** ist
  - eine Abweichung zwischen Ist-Verhalten (im Testlauf festgestellt) und Soll-Verhalten (in der Spezifikation gefordert) oder ein
  - nicht erfülltes, vom Kunden vorausgesetztes Qualitätskriterium
- Testen ist **destruktiv**: Es zeigt nicht die Korrektheit eines Programms, sondern seine Inkorrektheit

- Ein **Testdatum** ist ein Satz von Eingabe- und Zustandswerten für ein Softwareobjekt mit zugehörigen Ausgabe-Sollwerten
- Ein **Testfall** ist eine allgemeine Beschreibung eines Testdatums oder einer Folge von Testdaten für ein zu testendes Software-objekt
- Testfälle sollten auf systematische Weise gewählt werden, um eine gewisse Wahrscheinlichkeit zu bieten, Fehler aufzudecken
- Nach der Methoden der Testfall-Ermittlung unterscheidet man
  - Black-Box-Testen
  - White-Box-Testen

Black-Box-Test	White-Box-Test
• Testfälle gehen von der Spezifikation aus	• Testfälle ausgehend von der Struktur des Testobjekt
• Interna des Testobjekts sind bei der Ermittlung der Testfälle unbekannt	• Testfälle werden vom Entwickler beschrieben
• Testüberdeckung wird an Hand des spezifizierten Ein/Ausgabeverhaltens gemessen	• Testüberdeckung wird an Hand des Codes gemessen

Methoden des Black-Box-Test

- Äquivalenzklassenmethode
- Methode der Grenzwertanalyse
- Test von Zustandsautomaten

Methoden des White-Box-Test

- kontrollflussorientierte Verfahren
  - Anweisungsüberdeckungsverfahren
  - Pfadüberdeckungsverfahren
- datenflussorientierte Verfahren (*wird nicht behandelt*)

3.2.1 Methoden des Black-Box-Test

Äquivalenzklassenmethode (heuristisches Verfahren)

- Eine Äquivalenzklasse ist eine Menge von Eingabewerten, die auf ein Programm eine gleichartige Wirkung ausüben
- Es werden Äquivalenzklassen gültiger und ungültiger Werte gebildet, welche den Eingabebereich abdecken
- die Testfälle entsprechen(Kombinationen von) Äquivalenzklassen
- aus jeder Äquivalenzklasse wird mindestens ein Testdatum gewählt

Grenzwertanalyse

- basierend auf der Äquvalenzklassenmethode
- aus jeder Äquivalenzklasse werden Testdaten gewählt, welche Grenzwerte der Äquvalenzklassen abdecken

Äquivalenzklassenmethode: Beispiel 1

Funktion zur Bestimmung der Anzahl der Tage in einem Monat
Eingabe: int monat, int jahr

- monat: Äquivalenzklassen gültiger Werte

  $C_{m,31}$ = Monate mit 31 Tagen = {1,3,5,7,8,10,12}
  $C_{m,30}$ = Monate mit 30 Tagen = {4,6,9,11}
  $C_{m,feb}$ = {2}
  Ungültige Werte: Negative Zahlen, Zahlen > 12

- jahr: Äquivalenzklassen gültiger Werte

  $C_{j,schaltahr}$ = Menge der Schaltjahre
  $C_{j,nicht\text{-}schaltjahr}$ = Menge der Nicht-Schaltjahre
  Ungültige Werte: Negative Zahlen

Äquivalenzklassenmethode: Beispiel 1

Ableitung von Testfällen gültiger Eingaben aus den Äquivalenzklassen

Testfall			ausgewähltes Testdatum		
	Äquivalenzklasse	Ausgabe	monat	jahr	Ausgabe: Soll
$T_{1,1}$	$C_{m,31}$ und $C_{j,nicht\text{-}schaltjahr}$	31	7	1901	31
$T_{1,2}$	$C_{m,31}$ und $C_{j\ schaltjahr}$	31	7	1904	31
$T_{2,1}$	$C_{m,30}$ und $C_{j,nicht\text{-}schaltjahr}$	30	6	1901	30
$T_{2,2}$	$C_{m,30}$ und $C_{j\ schaltjahr}$	30	6	1904	30
$T_{3,1}$	$C_{m,feb}$ und $C_{j,nicht\text{-}schaltjahr}$	28	2	1901	28
$T_{3,2}$	$C_{m,feb}$ und $C_{j,\ schaltjahr}$	29	2	1904	29

Äquivalenzklassenmethode: Beispiel 2

Spezifikation (noch nicht ganz konsistent) zur Ableitung des technischen Eintrittsalters einer Person in einen Versicherungsvertrag:

Eingabe: vertragsbeginn, geburtsdatum	
Hilfsvariable diff_Monat := Monat(vertragsbeginn) – Monat (geburtsdatum) diff_Jahr := Jahr (vertragsbeginn) – Jahr (geburtsdatum)	
technisches_Eintrittsalter	**Bedingung**
Fehler	vertragsbeginn < geburtsdatum
diff_Jahr	vertragsbeginn >= geburtsdatum und -5 <= diff_Monat <= 6
diff_Jahr + 1	vertragsbeginn >= geburtsdatum und diff_Monat >= 6
diff_Jahr - 1	vertragsbeginn >= geburtsdatum und diff_Monat < - 5

Äquivalenzklassenmethode: Beispiel 2

Test -fall			Ausgewähltes Testdatum		
	Äquivalenzklasse	Ausgabe	Geburts- datum	Vertrags- beginn	Ausgabe : Soll
T1	1 Vertragsbeginn vor Geburtsdatum	Fehler	01.02.2001	01.01.2001	Fehler
T2	2 diff_Monat im Intervall [-5, 6]	diff_Jahr	01.06.1975	01.08.2001	26
T3	3 diff_Monat >= 6	diff_Jahr + 1	01.06.1975	01.12.2001	27
T4	4 diff_Monat < - 5	diff_Jahr - 1	01.10.1975	01.01.2001	25

Klasse 1 ist eine Klasse ungültiger Werte
Weitere ungültige Klassen: Tag, Monat, Jahr nicht im gültigen Werteberei

Grenzwertmethode: Beispiel 2

Ti-j : Testfall in Äquivalenzklasse i an der Grenze zu Klasse j

Testfall			Ausgewähltes Testdatum		
	Eingabe	Ausgabe	Geburts -datum	Vertrags -beginn	Ausgabe : Soll
T1-2	Vertragsbeginn 1 Tag vor Geburtsdatum	Fehler			
T2-1	Vertragsbeginn = Geburtsdatum	0			
T2-3	**diff_Monat = 6**	diff_Jahr			
T2-4	diff_Monat = - 5				
T3-2	**diff_Monat = 6**	diff_Jahr + 1			
T4-2	diff_Monat = - 6	diff_Jahr - 1			

Nebeneffekt: Es wurde eine Inkonsistenz in der Spezifikation gefunden

Test von Zustandsautomaten

- Testobjekte mit interessantem dynamischem Verhalten
- Voraussetzung: Zustandsautomat bzw. Ablaufdiagramm liegt als Spezifikation des Testobjekts vor, beispielsweise
  - Statecharts
  - Objektlebenszyklus aus OOA-Modellierung
  - Aktivitätendiagramme
- Ein Testfall beschreibt eine Folge von Ereignissen, die auf das Testobjekt ausgehend von seinem Anfangszustand angewendet werden.
- Für jeden Schritt dieser Folge wird als Soll der nächste erwartete Zustand und die erwartete Ausgabe festgelegt.

Test von Zustandsautomaten: Beispiel eines Parkscheinautomaten [Beispiel von H.Balzert]

Beispiel eines Testfalls

Anfangszustand: bereit

Ereignisse:

Karte eingeschoben

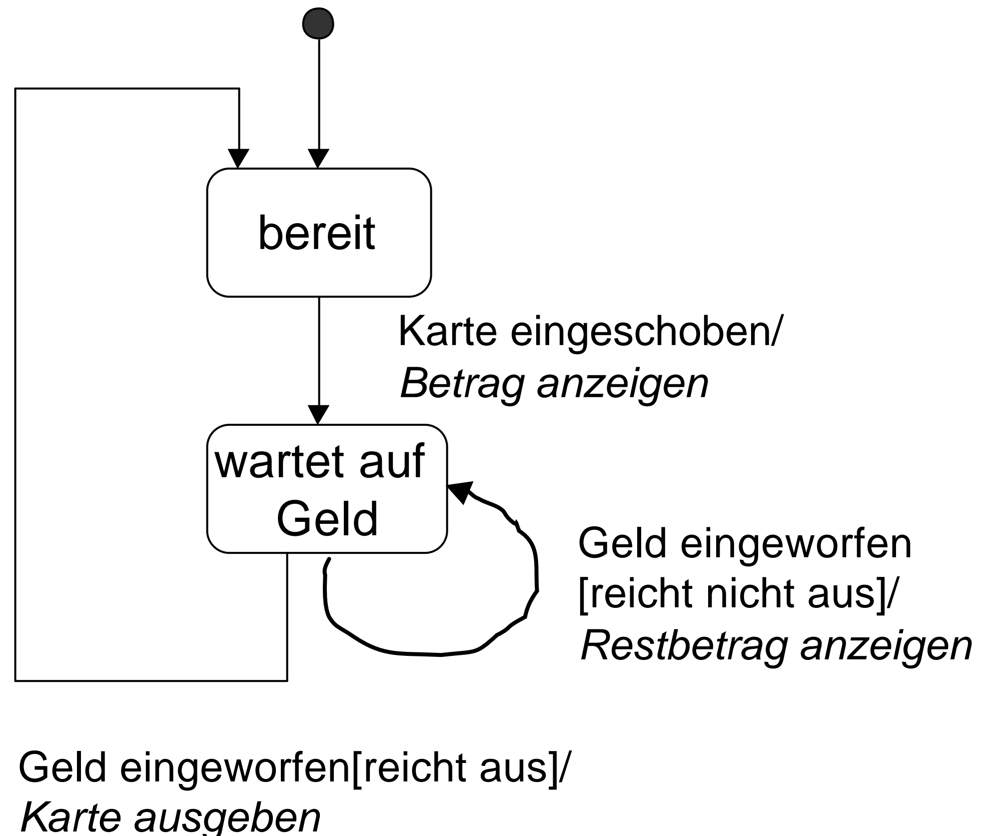
(*Nachfolgezustand Soll*: → wartet auf Geld)

Geld eingeworfen [reicht nicht aus]

(*Nachfolgezustand Soll*: → wartet auf Geld)

Geld eingeworfen [reicht aus]

(*Nachfolgezustand Soll*: →bereit)



Test von Zustandsautomaten

- Ziel: Durch Testfälle Überdeckung aller Zustandsübergänge
- Zum Soll/Ist-Vergleich beim Testlauf erforderlich:

  Anreicherung des zu testenden Programms um Code
  zur Verfolgung der durchlaufenen Zustände (bsp. Protokollierung von Zustandsattributen)

Methoden des Black-Box-Test alleine sind nicht ausreichend, da die Spezifikation ein höheres Abstraktionsniveau besitzt wie die Implementierung

- nicht alle Elemente einer funktionalen Äquivalenzklasse werden in der Implementierung „intern“ gleich behandelt
- in der Implementierung kann ein Zustand im Automaten mehreren internen Zuständen entsprechen

3.2.2 Methoden des White-Box-Test

Kontrollflussorientierte Testverfahren

orientieren sich am Kontrollflußgraphen des Programms

1. Anweisungsüberdeckungsverfahren (C0-Test)

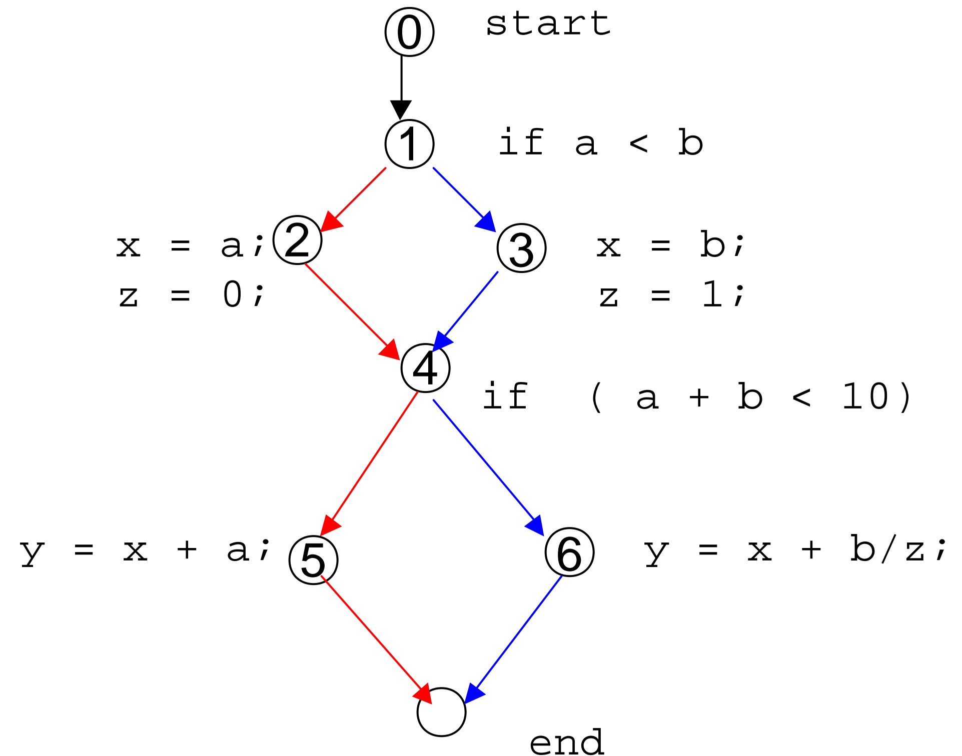
- Ziel: Alle Anweisungen des Testobjekts werden ausgeführt
- Es gibt Werkzeuge zur Messung der C0-Überdeckung messen
- Schwächen des C0-Verfahrens
  - es müssen nicht alle Wege, die zu einer Anweisung führen überprüft werden: einer genügt
  - unzureichend für den Test von Schleifen

Beispiel einer fehlerhaften Routine

[Beispiel von J. Coldewey]

C0- Überdeckung bei Testfällen
T1: (a < b und a + b < 10)
und
T2: (a >= b und a + b >= 10)

Fehler im Pfad
0; 1; 2; 4; 6
wird dabei nicht entdeckt



2. Pfadüberdeckungsverfahren

- Ziel: Alle möglichen Pfade des Kontrollflußgraphen werden durchlaufen
- nicht realistisch für while-Programme
- boundary-interior Pfadtest: die Anzahl der Wiederholungen in Schleifen wird eingeschränkt
- Keine Unterstützung dieser Verfahren durch marktgängige Werkzeuge

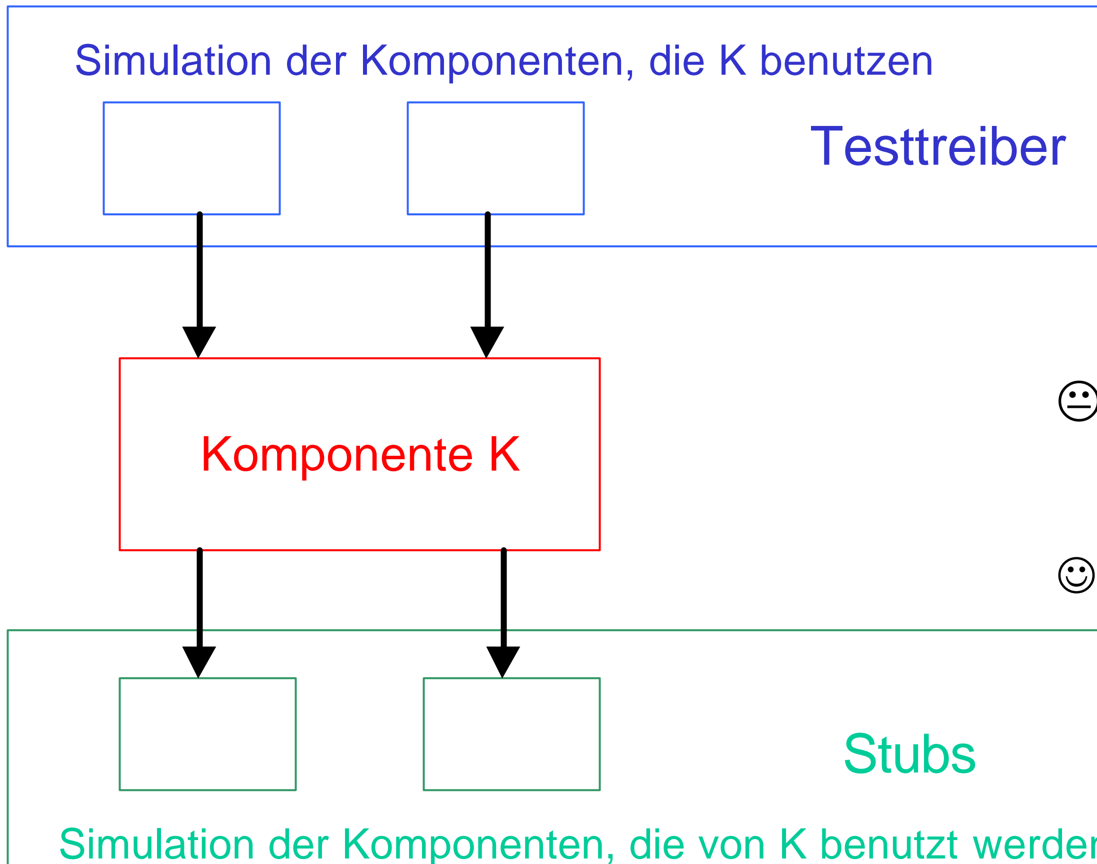
4. Tests im Software-Entwicklungsprozess

- **Unit-Test oder Modultest**: Testen einer einzelnen fertiggestellten Systemkomponente
- **Integrationstest**: Test der Schnittstellen und des Zusammen-wirkens mehrerer Systemkomponenten
- **Systemtest:** Test des vollständig integrierten Systems in der Zielumgebung gegen die Anforderungen

  nach Korrekturen: **Regressionstests**

- **Abnahmetest:** Systemtest des Auftraggebers in der realen Einsatzumgebung und mit echten Daten

Testumgebung für den Unit-Test



☹ Aufwand für Erstellung von Treibern und Stubs

☺ - Lokalisierung von Fehlern
- Paralleles Testen verschiedener Komponente

Kombination von Black-Box-Test und White-Box-Test

- Testerin legt Black-Box-Testfälle an Hand der Spezifikation der Komponente fest (destruktives Vorgehen!)
- Zu den Testfällen werden Testdaten bereitgestellt
- Entwickler führt Testfälle durch
- C0-Überdeckung wird gemessen und analysiert
- Entwickler reichert Testfälle aus dem Funktionstest an um
  - Testfälle für Fehlerbehandlung
  - Testfälle für interne algorithmische Lösungen
- C0-Überdeckung aller Testfälle wird gemessen
- Überdeckungstest ist beendet, wenn gewünschte Rate erreicht ist.
- Bei Fehlern: Korrektur und **Regressionstest**

Regressionstest

Wiederholung eines schon durchgeführten Tests nach einer Korrektur zur Überprüfung, dass nur die erwünschten Änderungen auftreten

Beispiel

technisches Eintrittsalter, Ersttest mit Programm Version V1.0

Testfall	Geburts-datum	Vertrags-beginn	Ausgabe: Soll	Test mit V1.0	Ausgabe: Ist	Erg
T1-2	02.01.2001	01.01.2001	Fehler		Fehler	ok
T2-1	01.01.2001	01.01.2001	0		0	ok
T2-3	01.06.1975	01.12.2001	26		27	nok
T2-4	01.06.1975	01.01.2001	26		26	ok
T3-2	01.05.1975	01.12.2001	27		27	ok
T4-2	01.07.1975	01.01.2001	25		25	ok

Beispiel (fortgesetzt):

Fehlerkorrektur in Version V1.1 und anschließender Regressionstest von V1.1 bzgl. der Vorgängerversion V1.0

Testfall	Test mit V1.0	Ist	Erg
T1-2		Fehler	ok
T2-1		0	ok
T2-3		27	nok
T2-4		26	ok
T3-2		27	ok
T4-2		25	ok

Testfall	Test mit V1.1	Ist	Erg
T1-2		Fehler	**ok**
T2-1		0	**ok**
T2-3		26	**prüfen**
T2-4		26	**ok**
T3-2		25	nok
T4-2		25	**ok**

4.2 Integrationstest

Voraussetzung:

Unit-Test der beteiligten Komponenten ist erfolgt

Aufgabe des Integrationstests:

Tests der Schnittstellen und des Zusammenwirkens der Komponenten ⇒ **White-Box-Test**

Integrationsstrategien

☺ **inkrementell**
sukzessive Integration nach einer gewählten Reihenfolge

😐 **nicht-inkrementell**
big-bang oder geschäftsprozessorientiert

Inkrementelle Integrationsstrategien

- *bottom-up*: ausgehend von den untersten Komponenten in der Nutzungshierarchie sukzessive nach oben
  $\Rightarrow$ *im bottom-up Test keine Stubs*
- *top-down:* ausgehend von den obersten Komponenten in der Nutzungshierarchie sukzessive nach unten
  $\Rightarrow$ *im top-down Test keineTesttreiber*
- *inside-out (sandwich):* von der Mitte nach oben und unten

4.3 Systemtest

Voraussetzung

Der Integrationstest für das vollständig integrierte System ist abgeschlossen

Aufgaben des Systemtests

Black-Box-Test des fertigen Systems gegen die festgelegten Qualitätsziele, beinhaltet

- Funktionstest
- Leistungstest
- Lasttest
- Stresstest
- Benutzbarkeitstest
- Sicherheitstest
- Interoperabilitätstest

5. Prinzipien systematischen Testens

Tests rechtzeitig planen, spezifizieren und implementieren

- Tests planen
  - Teststrategie
  - Testorganisation
  - Testumgebung und Testwerkzeuge
  - Testdokumentation
- Tests spezifizieren
  - Definition der Testdurchführung
  - Beschreibung der Testfälle
- Tests implementieren
  - Aufbau der Testumgebung
  - Testtreiber, Stubs, Testdaten
- Tests durchführen

Tests wiederholbar machen

- Regressionstests sind erforderlich
  - nach jeder Fehlerkorrektur und jeder Realisierung eines Änderungswunsches
  - in der Entwicklung und in der Wartung
- Regressionstests dürfen daher mit nicht viel Aufwand verbunden sein
- Erstrebenswert: Automatisierung von Regressionstests (Test auf Knopfdruck)
  - automatische Durchführung der Testfälle
  - automatischer Vergleich von Testausgaben

Fazit

- Ein Test muss geplant sein
- Ein Test muss dokumentiert werden
- Ein Test muss wiederholbar sein
- Ein Test muss möglichst vollständig, aber auch bezahlbar sein

Literatur

- Helmut Balzert: Lehrbuch der Software-Technik, Band 2, Spektrum Akademischer Verlag, 1998.
- Bernd Bruegge, Allen H. Dutoit: Object-Oriented Software Engineering, Chapter 9, Prentice Hall, 2000.
- Jens Coldewey: Reviews reviewed, Objekt Spektrum Aug.-Okt. 2000
- Michael Fagan: Advances in Software Inspection, IEEE Transactions on Software Engineering, SE-12, July 1986, S. 744 - 751
- Andreas Mieth: Qualitätsmanagement, in:
  Peter Brössler, Johannes Siedersleben (Herausgeber): Softwaretechnik, Praxiswissen für Software-Ingenieure, Hanser 2000
- Ernest Wallmüller: Software-Qualitätssicherung in der Praxis, Hanser 1990
- Internet-Seiten mit Testbegriffen:
  - Fachgruppe 2.1.7 der Gesellschaft für Informatik (GI) http://www.fbe.hs-bremen.de/spillner/begriffe/home.html
  - Glossar der Firma Imbus: http://www.imbus.de/glossar.html