Software Qualitätssicherung
und
Projektmanagement
- Lernskript -

Hinweis

Bei dem vorliegenden Dokument handelt es sich lediglich um eine *persönliche* Zusammenfassung, wobei Formulierungen teilweise aus dem Skript übernommen, oder aber auch persönlich erstellt wurden. Inhaltliche Gewichtungen oder Interpretationen stimmen nicht unbedingt mit den Inhalten und Aussagen der Vorlesungen überein.
Daher ist das vorliegende Lernskript lediglich als Hilfestellung bei der Strukturierung der Vorlesungsinhalte zu verstehen. ***Es ersetzt keinesfalls die Vorlesung oder die Bearbeitung der Skripte und erhebt in keinster Weise Anspruch auf eine der genannten Eigenschaften sowie Vollständigkeit und Korrektheit!***

Trotz allem hoffe ich, mit diesem Skript ein wenig Unterstützung beim Erarbeiten der Veranstaltungsinhalte oder einer evtl. Prüfungsvorbereitung geben zu können.
Für Verbesserungsvorschläge und Korrekturen bin ich jederzeit dankbar.

28. Januar 2005

Vorlesungsbezug:

- Software Qualitätssicherung und Projektmanagement, Prof. Lichter Sommersemester 2004

Inhaltsverzeichnis

1. Grundlagen der Softwarequalitätssicherung

Die Softwarequalitätssicherung hat mit allen Bereichen der Softwareentwicklung zu tun und sollte idealerweise in jedem Teilbereich Einfluß besitzen: Im **Projektmanagement**, welches ein Projekt initiiert, plant, kontrolliert und abschließt, im **Konfigurationsmanagement**, welches eng mit dem Projektmanagement und der **Systemerstellung** verzahnt ist. Aus dem Konfigurationsmanagement fließen Produkte in die Qualitätssicherung ein. Die Ergebnisse der Qualitätssicherung fließen wiederum in die Systemerstellung ein. Die dort entwickelten Produkte und Änderungen zu diesen werden im Konfigurationsmanagement verwaltet.

Der Begriff „*Qualität*":
Qualität ist die Gesamtheit von Merkmalen einer Einheit bezüglich ihrer Eignung, festgelegte und vorausgesetzte Erfordernisse zu erfüllen. Eine Einheit kann ein Produkt eine Tätigkeit, ein Prozeß, ein System, eine Person, eine Organisation, etc. sein.
Die Ziele der Qualität können *explizit* festgelegt oder implizit durch gemeinsame Vorstellungen der Beteiligten gegeben werden.
Allerdings ist Qualität **kein absolutes Maß** für die Güte einer Einheit und ist immer **relativ** zu den geforderten Eigenschaften.
→ **Qualität entsteht nicht von selbst, d.h. sie muß definiert** werden.

Das ***Qualitätsmanagement*** ist eine unternehmerische Aufgabe, welche die ***Qualitätspolitik***, **Ziele** und **Verantwortlichkeiten** festlegen sowie diese durch Mittel wie
- Qualitätsplanung
- Qualitätslenkung
- Qualitätssicherung und –Verbesserung

Im Rahmen des **Qualitätsmanagementsystems** verwirklichen.

Heute heißt Qualitätssicherung Qualitätsmanagement, die Qualitätssicherung ist beschränkt auf die Qualitätsmanagement-Darlegung.

Qualitätspolitik
Der Stellwert der Qualität im Handeln und den daraus folgenden Auswirkungen muß in jedem Unternehmen festgelegt werden. Qualität kann auch ins Zentrum des Handelns gestellt werden. **Totales Qualitätsmanagement** macht Qualität zum Unternehmensprinzip; Kundenzufriedenheit ist oberstes Unternehmensziel. Übrige Ziele werden von diesem Ziel abgeleitet.

Qualitätsplanung
Qualität ist nie absolut, sondern immer auf einen Verwendungszweck bezogen (kein Sinn, „irgendeine Qualität" zu erzeugen).
→ Bestimmung von ***Qualitätszielen*** in Form von quantifizierten ***Qualitätsanforderungen***.
Das heißt für Software: *Es ist kein Qualitätsmanagement ohen eine saubere, quantifizierte Spezifikation der Anforderungen möglich.*

Qualitätsprüfungen und –lenkung: Analytische und konstruktive Arbeitstechniken und Tätigkeiten zur Erfüllung der Qualitätsanforderungen.

Qualitätsmanagement-Darlegung (QS): Tätigkeiten zur Schaffung von Vertrauen, daß die Qualitätsanforderungen erfüllt werden. (Q-Berichte, Prüfberichte, Audits).

Qualitätsverbesserung: Maßnahmen zur Erhöhung von Effektivität und Effizienz der qualitätsbezogenen Tätigkeiten.

Qualitätsmanagementsystem: Struktur, Verantwortlichkeiten und Mittel zur Verwirklichung des Qualitätsmanagements.

In der Entwicklung kann nichts gemessen werden, da keine Massenherstellung gleichartiger Komponenten stattfindet. So münden die Prüfungen meist in Individualprüfungen. Eine Rückkopplung auf den Prozeß ist schwierig.

Grundsätze zum Qualitätsmanagement:

- Produktverantwortung und Qualitätsverantwortung sind untrennbar miteinander verbunden
- Die Ermittlung von Qualität (Messung, Prüfung) kann an Spezialisten (Q-Ingenieure) delegiert werden (Q-Organisation muß etabliert sein)
- Qualität muß erzeugt werden, sie ist nicht hineinprüfbar
- Alle Beteiligten müssen über die Qualität ihrer Arbeit informiert sein (Verantwortlichkeiten müssen definiert sein)
- Ein schlechter Entwicklungsprozeß fördert die Entstehung schlechter Produkte (Verbesserung des Entwicklungsprozesses muß kontinuierlich sein)

Merkmale der SW-Qualitätssicherung

- Nur Entwicklung, keine Produktion
- Keine tradierten Standards
- Keine allgemein anerkannten Qualitätsbegriffe
- Immaterielle Produkte sind schwierig zu messen und zu prüfen
- Rückkopplung; Wird heute nur für Nebenprobleme (z.B. Gliederung von Dokumenten), nicht aber für zentrale Qualitätsziele (z.B. Benutzerfreundlichkeit) beherrscht

Merkmale der Prozeß-Qualität (hoch = gut)

- Inverse Dauer
- Inverser Aufwand
- Termin-Einhaltung
- Aufwandseinhaltung
- Prozeß-Transparenz
- Baustein-Gewinn
- Know-How-Gewinn
- Projekt-Klima

Merkmale der Produkt-Qualität

- Spezifikations-Vollständigkeit
- Lokalität
- Testbarkeit
- Strukturiertheit
- Simplizität
- Knappheit (wenig Redundanz)
- Lesbarkeit
- Geräteunabhängigkeit
- Abgeschlossenheit
- Korrektheit

- Ausfallsicherheit
- Genauigkeit
- Effizienz
- Sparsamkeit (kaum mehr Speicherplatz und andere Betriebsmittel benötigt als minimal erforderlich)
- Leistungsvollständigkeit
- Konsistenz (ähnliches Verhalten in ähnlichen Situationen)
- Handbuch-Vollständigkeit
- Einfachheit

Qualitätsfaktoren
- Korrektheit
- Zuverlässigkeit
- Effizienz
- Integrität (Verhinderung von unberechtigtem Zugriff)
- Verwendbarkeit (Leichtigkeit der Erlernung der Bedienbarkeit der Software und der Interpretierbarkeit ihrer Ergebnisse)
- Wartbarkeit (Leichtigkeit, System zu verändern; Fehlerbeseitigung, Anpassung an neue Umgebungen)
- Flexibilität
- Testbarkeit
- Portabilität
- Wiederverwendbarkeit (Ein Stück Software soll in anderen Software-Projekten wiederverwendet werden)
- Verknüpfbarkeit (Leichtigkeit des Informationsaustauschs zwischen verschiedenen Systemen und der Benutzung der ausgetauschten Informationen)

Mangel: *Die Nichterfüllung einer beabsichtigten Forderung oder einer angemessenen Erwartung für den Gebrauch eines Software-Produkts.* In diesem Fall entspricht die Anforderungsdefinition nicht den Forderungen des Auftraggebers (Beispiel: Benutzungsfreundlichkeit).

Fehler: *Die Nichterfüllung einer festgelegten Forderung, insbesondere eines Qualitäts- und Zuverlässigkeitsmerkmals. Ob ein Leistungsmangel ein Fehler ist, hängt also davon ab, ob die erforderliche Leistung in der Anforderungsdefinition exakt festgelegt ist.*

Fehler und Fehlerentstehung:
Eine Person begeht einen *Irrtum (mistake).*
Als mögliche Folge davon enthält die Software einen *Defekt (defect,fault).*
Wird der Defekt durch Prüfen der Software gefunden, so ergibt das einen *Befund (finding).*
Bei der Ausführung von Software mit einem Defekt kommt es zu einem ***Fehler*** *(error)* (die tatsächlichen Ergebnisse weichen von den erwarteten / richtigen ab).
Dies kann zu einem *Ausfall (failure)* eines software-basierten System führen.

Umgangssprachlich wird in allen fünf Fällen von einem „Fehler“ gesprochen.
Im Qualitätsmanagement ist die genaue Unterscheidung oft wichtig, denn
- ein Defekt hat häufig mehrere Fehler zur Folge
- Ein Defekt führt nicht immer zu einem Fehler
- Nicht jeder Fehler hat einen Defekt als Ursache
- Nicht jeder Fehler hat einen Ausfall zur Folge

Defekte → Fehler → Ausfälle

Defekte werden auch begangene Fehler oder Fehlerursachen genannt. Fehler werden manchmal präziser als gefundene, erkannte oder festgestellte Fehler bezeichnet.

Fehlerarten:
- Funktionsfehler (Funktionalität, Datenstrukturen)
- Schnittstellenfehler
- Algorithmusfehler
- Zuweisungsfehler
- Abfragefehler (Programmlogik)
- Synchronisations- und Zeitfehler (Zugriff auf gemeinsame Ressourcen)
- Konfigurationsfehler (Verwendung von falschen Versionen in einer Konfiguration)
- Dokumentationsfehler (Abweichung von der Realisierung)

Fehlerentstehung:
Fehler können in jeder Phase / bei jeder Tätigkeit der SW-Entwicklung entstehen. Es gibt keine Gleichverteilung. Je später Fehler in der Entwicklung entdeckt werden desto teuer wird ihre Behebung. Dies gilt gerade und besonders für Fehler in der Anforderungsspezifikation. Entwurfs- und Codierungsfehler sind um rund die Hälfte weniger aufwendig.

Einsatz von QS-Maßnahmen:
Mit Hilfe von *Analytische Maßnahmen (Software-Prüfung, Metriken)* sollen Zwischen- und Endergebnisse überprüft und erkannte Fehler korrigiert werden. Weiterhin soll eine Überprüfung der Einhaltung des geplanten Entwicklungsprozesses stattfinden.

Konstruktive und *organisatorische Maßnahmen (Vorgehensmodelle, Datenkapselungen, Hochsprachen bzw. Verantwortung, Richtlinien, Audits)* definieren fehlerverhindernde oder fehlervermeidende Prozesse. Prüf- und Korrekturverfahren werden in die Prozesse integriert. Die daraus gewonnenen Ergebnisse werden zur Verbesserung und Optimierung der Prozesse verwendet.

Hinweis: Eine systematische, ingenieurmäßige Vorgehensweise, welche die Erreichung gegebener Qualitätsanforderungen garantiert, gibt es für Software bis heute nicht. Einsatz konstruktiver Maßnahmen um generelles Qualitätsniveau zu heben. Mittel der Praxis: Rigorose Qualitätsprüfung.

Validierung („Tun wir das Richtige?"): *Prozeß der Beurteilung eines Systems oder eine Komponente während oder am Ende des Entwicklungsprozesses. Ziel: Feststellung, ob spezifizierte Anforderungen erfüllt sind.*

Verifikation („Tun wir es richtig?"):
- *Prozeß der Beurteilung eines Systems oder einer Komponente mit dem Ziel, festzustellen, ob die Resultate einer gegebnen Entwicklungsphase den Vorgaben für diese Phase entsprechen.*
- *Formaler Beweis der Korrektheit des Programms.*

Die Benutzererwartungen müssen in allen Entwicklungsphasen validiert werden, die Verifikation läuft stufenweise ab, bis der resultierende Code (das Programm) wieder mit der Anforderungsspezifikation verglichen wird.

Grundsätze der Prüfung von Software:

- Nur gegen Vorgaben prüfen
- Prüfung planen
- Prüfverfahren müssen wohldefiniert sein und reproduzierbare Ergebnisse liefern
- Prüfergebnisse müssen dokumentiert werden
- Beim Prüfen erkannte Fehler müssen anschließend korrigiert werden (verursachende Defekte erkennen und beheben)

1.1. Exkurs: Anforderungsspezifikationen

Die Anforderungsspezifikation dokumentiert die wesentlichen Anforderungen (= Minimalbedingungen hinsichtlich der Funktion und Qualität) an eine Software und ihre Schnittstellen in ***präziser, vollständiger*** und ***überprüfbarer*** Weise.
Die Spezifikation sagt, **was** das System leisten soll, nicht **wie** es funktionieren soll (Lösungsunabhängigkeit).
Das ***Pflichtenheft*** ist die vertragliche Ausgestaltung der Spezifikation.

Nutzen der Spezifikation:
Sie ist der Dreh- und Angelpunkt der Software-Entwicklung und erforderlich für:

- Abstimmung mit dem Kunden
- Entwurf
- Benutzerhandbuch
- Testvorbereitung
- Abnahme
- Wiederverwendung und Reimplementierung

Konsequenzen bei Fehlen der Spezifikation:

- ungeklärte Anforderungen
- keine Vorgaben für Entwerfer
- fehlende Basis für Benutzerhandbuch
- Test kann nicht vorbereitet werden
- Grundlage für Abnahme und Nachforderungen fehlt
- Wiederverwendung und Reimplementierung stark erschwert

Anforderungen sollten folgenden Kriterien genügen:

- zutreffend (korrekt)
- vollständig
- widerspruchsfrei (konsistent)
- umsetzbar
- neutral (bzgl. der Realisierung)
- testbar
- notwendig (tatsächliche Anforderungen des Kunden sollen aufgenommen werden)
- bewertbar (größere Entwicklungen: Priorisierung der Anforderungen nach Fach, Wichtigkeit und Dringlichkeit)
- Darstellung leicht verständlich und präzise
- In der Form leicht erstellbar, leicht verwaltbar, leicht verfolgbar

Im Normalfall bilden Anforderungen an ein Software-System kein formales Modell der fachlichen Logik. Typischerweise sind sie unvollständig, mehrdeutig und widersprüchlich → Wahl geeigneter Notation notwendig.

Natürliche Sprache:
Vorteile: Jeder beherrscht sie → sie muß nicht erst erlernt werden
Nachteile: vage Aussagen, die zudem unterschiedlich interpretiert werde können. Außerdem können sie unvollständig und widersprüchlich sein.

Formale Sprache:
Vorteil: definierte Semantik für Aussagen, kein Interpretationsspielraum. Automatische Prüfung möglich.
Nachteile: Nur verständlich für Spezialisten

RE-Notationen decken nur ganz spezielle Aspekte ab. Nicht-funktionale Anforderungen werden i. d .R. natürlichsprachlich formuliert. Die objektorientierte Analyse kann als ein kombinierter Formalismus betrachtet werden.

Hinweise:
- sprachlich exakte Formulierung
- Quantifizierung der Anforderungen
- Kennzeichnen der Anforderungen
- Verwenden von Grafiken (bspw. um Anforderungen zu erläutern
- Anlegen eines Begriffslexikons
- Wenn Informationen nicht verfügbar sind, diese Stellen kennzeichnen
- Nicht relevante Abschnitte im Musterdokumenten sollen *gekennzeichnet* werden, *nicht gelöscht!*

2. Analytische Maßnahmen

2.1. Statische Prüfung

2.1.1. Linguistischer Ansatz

Um aus den Anforderungen auf eindeutige Art und Weise eine lauffähige Lösung entwickeln zu können, muß das Ergebnis der Analyse formal = vollständig + widerspruchsfrei + eindeutig sein. Normalerweise ist dies nicht der Fall.
Fragestellung: Wie gelangt man an das, was der Urheber von Anforderungen an ein zukünftiges System meinte, als er die Anforderungen erzeugte?
(vgl. Beispiele in Vorlesung Nr. 03)

Ausgehend davon, daß Regeln existieren, nach denen Menschen offenbar vorgehen, wenn sie sich natürlichsprachlich äußern, sollen Defekte in natürlichsprachlich definierten Anforderungen an Softwaresysteme systematisch aufgespürt werden (analytische QS-Maßnahme). Hierzu soll eine Checkliste bereitgestellt werden, die bei der Erstellung und Prüfung natürlichsprachlicher Anforderungen eingesetzt werden kann.

Transformationen, die Menschen einsetzen, um ihre Wahrnehmung in einen Teil ihres persönlichen Weltbildes umzugestalten, können beschrieben werden. Es werden also persönliche Modelle der Umwelt erstellt. Aus den sprachlichen *Oberflächenstrukturen* muß der Analytiker nun die *Tiefenstrukturen* zurücktransformieren.

Transformationskategorien:

- ***Tilgung***
  - Implizite Annahmen
  - Unvollständig spezifizierte Prozeßwörter
  - Unvollständige Vergleiche und Steigerungen
  - Modaloperatoren der Möglichkeit
  - Modaloperatoren der Notwendigkeit
- ***Generalisierung***
  - Universalquantoren
  - Unvollständig spezifizierte Bedingungen
  - Substantive ohne Bezugsindex
- ***Verzerrung***
  - Nominalisierung
  - Funktionsverbgefüge

2.1.2. Reviewtechnik

Prüfungen liefern implizit eine Definition der Qualitätskriterien. Sie zeigen damit den Entwicklern, welchen konkreten Anforderungen ihre Produkte genügen sollten, denn oft ist die Spezifikation nur unzureichend. Prüfungen zeigen weiterhin kollektive und individuelle Schwächen der Prüflinge (Rückkopplung), die Qualität der Prüflinge erhöhen sie allerdings nicht. Extrem gute und extrem schlechte Prüflinge können allerdings entdeckt werden. Die Erwartung einer Prüfung beeinflußt auch das Verhalten der Entwickler, denn diese wollen Software liefern, die gute Prüfungsresultate erzielt. Auf diesem Weg wird also auch die Qualität des Produkts indirekt erhöht.

Reviews sind analytische Maßnahmen im Rahmen der Qualitätssicherung, bei der ein Prüfling (Dokument) *statisch* und ohne Rechnerunterstützung von Menschen geprüft wird.
Geprüft wird gegenüber Vorgaben und Richtlinien mit dem Ziel, Fehler und Schwächen des Prüflings aufzuzeigen, aber auch positive Merkmale zu würdigen.
Bei solchen Inspektionen sollen Produkte und Prozesse verbessert werden. Im Anschluß daran folgen Tests sofern sie durchführbar sind.

Allgemeine Regeln für Reviews:

- Sitzungsdauer max. 2 Stunden
- Moderator kann Sitzung absagen oder abbrechen
- Das Dokument und nicht der Autor stehen zur Diskussion
- Moderator darf nicht gleichzeitig auch Experte sein
- Keine Diskussion von Stilfragen
- Keine Diskussion von Lösungen
- Experten müssen Befunde angemessen präsentieren
- Der Konsens der Experten zu einem Befund ist laufend zu dokumentieren
- Befunde sind nicht als Anweisungen an den Autor zu formulieren
- Jeder Befund ist in kritische, nicht-kritische Fehler oder auch „gut“ zu gewichten
- Das Review-Team muß eine Empfehlung abgeben
- Alle Review-Teilnehmer unterschreiben das Protokoll

Allgemeine Hinweise:

- Einplanung von Reviews während der Entwicklungszeit
- Genügend Zeit für Vorbereitung muß zur Verfügung stehen
- Verwende Checklisten

- Mache Verbesserungsvorschläge auch für das Review-Verfahren (auch:Dritte Stunde)
- Prüfe unter der Annahme, daß alle Regeln befolgt wurden
- Prüfe auf Inkonsistenzen

Klassifikation von Inspection Reviews:

- *Technisches Review*
  Beteiligte: Moderator(Leitung, Planung des Reviews – neutrale Rolle), Protokollführer (Dokumentation der Ergebnisse → Erstellung des Review Berichts), Gutachter (2-5, Expertenbeurteilung), Autor (Bereitstellung des Prüflings, verantwortlich für Prüfbereitschaft des Prüflings). Außerhalb des Reviews: Manager (keine Teilnahme am Review, aber Einplanung eines solchen. Entscheidung über Nachbearbeitung oder Freigabe aufgrund des Reviewberichts und der Bewertung / Empfehlung des Review - Teams)
- *Peer Review*
- Gutachter gehen in Klausur → Ergebnis: Gutachten. Keine allgemeinen Verfahrensregeln → Selbstbestimmung des Teams. Moderator leitet das Team und organisiert die Arbeit. Ad hoc – Zusammenstellung des Teams oder permanente Einrichtung („professionelle Peers“).
- *Structured Walkthrough*
  Code Review ist schwierig in kleinen Entwicklungsumgebungen. Weniger streng als Reviews. Autor leitet und moderiert die Sitzung. Vorstellung mit und ohne Vorbereitung. Trotz Bewahrung vieler Vorteile eines Reviews auch Nachteile: Kein Moderator, keine Regeln, keine Kontrolle, Ermessen des Autors, Zahl der gefundenen Fehler ist weitaus geringer als bei geregelten Reviews.
- *Stellungsnahme*
  Autor entscheidet, wer und wie viele Personen die Dokumente durchsehen. Er sammelt Feedback zur Überarbeitung. Nachteile: ähnlich zum Walkthrough: „Zwischen-Tür-und-Angel-Tests“, Gutdünken des Autors
  - *Schreibtischtest*
  Entwickler führt die Durchsicht allein durch → selbstverständliche und obligatorische Prüfung!
- *Design & Code Inspection*
  - Einführungssitzung mit Vorstellung des Prüflings und seines Umfeldes
  - Gutachter geben zu Beginn der Sitzung ihre Notizen an den Moderator ab
  - Vorleser für Prüfling → Erfassung der Befunde zum vorgelesenen Teil
  - Review-Team ist Entscheidungsinstanz für die Freigabe des geprüften Arbeitsergebnisses
  - Stärkere Formalisierung des Review-Berichts (Fehlergewichtung mit „major“ und „minor“)
  - Es werden mehr Daten erfaßt und viele Kennzahlen ermittelt

  Code Inspektionen können automatisiert werden – Code soll gegen Programmierrichtlinien geprüft werden. Weiterhin Prüfung, ob Code den Entwurfsdokumenten entspricht. Autor darf nicht Prüfer sein. Prüfe, um Programmierfehler zu entdecken.

Reviews stellen eine sehr effiziente Methode zur Fehlererkennung dar. Unter Verwendung von Review-*Werkzeugen* können Metriken berechnet werden und Hinweise auf komplexe, schlecht portable Teile und Verstöße gegen Programmierrichtlinien gefunden werden. Wenn solche Werkzeuge verwendet werden,, können sich menschliche Prüfer auf Aspekte konzentrieren, die das Werkzeug nicht prüfen kann → Prüfungseffizienz wird erheblich verbessert. Beispiele sind QA/C++ oder Logiscope.

2.2. Testen (klassisch)

2.2.1. Grundlagen

Testen ist der Prozeß, ein Programm mit der Absicht auszuführen, Fehler zu finden. Nach einem sorgfältigen Test, steigt die Wahrscheinlichkeit, daß das Programm sich auch in nicht getesteten Fällen wunschgemäß verhält. Unter Testen versteht man die Ausführung eines Programms unter Bedingungen, für die das korrekte Ergebnis bekannt ist und mit dem des Programms verglichen werden kann → Stimmen beide nicht überein, so liegt ein Fehler vor. Somit ist das Ziel des Testens, Fehler zu entdecken. Damit ist ein Test erfolgreich, wenn er einen Fehler gefunden hat. ***Aber:*** Ein erfolgloser Test ist niemals ein Beweis für ein korrektes Programm – es wurden lediglich keine Fehler gefunden!
→ *Die Korrektheit eines Programms kann durch Testen (außer in trivialen Fällen) nicht bewiesen werden. Ansonsten müßten alle möglichen Eingaben getestet werden → Komplexität nicht beherrschbar.*
Test-Orakel (meist Menschen) bestimmen Sollergebnisse. Diese sind notwendig, um Fehler zu erkennen. Das Orakel darf nicht das Programm sein, daß getestet wird, idealerweise werden sie generiert.
Problem: *Auch Sollwerte können fehlerhaft sein. Weichen Ist- und Sollwert voneinander ab, sollte zuerst der Sollwert geprüft werden.*

Tests
- Vorteile:
- natürliches Prüfverfahren
- reproduzierbar
- investierter Aufwand mehrfach nutzbar
- Zielumgebung wird mitgeprüft
- Systemverhalten wird sichtbar gemacht

- Nachteile:
- nicht alle Programmeigenschaften sind testbar
- nicht alle Anwendungssituationen sind nachbildbar
- Tests zeigen nicht die Fehlerursache

Klassifikation:
- Laufversuch (Übersetzen und Binden – Start)
- Wegwerf-Test (jemand führt das Programm aus und macht einige Tests)
- Systematischer Test (Ableitung von Testdaten aus Spezifikationen, Ausführung des Programms, Resultatsvergleich, Dokumentation aller Daten)

Prinzipien des Testens:
- Vollständiges Testen unmöglich (nur Stichproben)
- Kreative und anspruchsvolle Tätigkeit
- Tests müssen geplant sein
- Tests erfordern Unabhängigkeit
- Zu jedem Testfall gehört ein Soll-Resultat
- Entwicklungsorganisation sollte ihre Produkte nicht selber testen
- Überprüfe Testergebnisse gründlich
- Testfälle müssen auch für ungültige und unerwartete Eingaben definiert werden
- Vermeide Wegwerftestfälle
- Ein erfolgreicher Testfall ist dadurch gekennzeichnet, daß er einen bisher unbekannten Fehler entdeckt
- Ein Test ist nur so gut wie seine Testfälle

Testebenen:

- *Modultest:* Prüft gegen den Modulentwurf
- *Integrationstest:* prüft die integrierten Module gegen den Modulentwurf
- *Systemtest:* Prüft das Gesamtsystem gegen die Spezifikation
- *Abnahmetest:* prüft die Übereinstimmung des Gesamtsystems mit den Kundenerwartungen.

Testauswahl – Kriterien:

Für ein gegebenes Programm P mit seiner Spezifikation S, definiert ein Testauswahlkriterium die Bedingungen, die die Menger der Testfälle erfüllen müssen. (z.B.: „Alle Anweisungen von P müssen einmal durchlaufen werden!“)
Ein Generieren der Testfälle aufgrund der Basis eines Testfall-Kriteriums ist im allgemeinen nicht möglich.
Ein Testauswahlkriterium ist *zuverlässig*, wenn alle Mengen von Testfällen, die dem Kriterium genügen, dieselben Fehler entdecken.
Ein Testauswahlkriterium wird als *gültig* bezeichnet, wenn es für jeden Fehler eine Menge von Testfällen gibt, die dem Kriterium genügt, und den Fehler entdeckt.
Folgerung: Ist ein Testauswahlkriterium sowohl zuverlässig als auch gültig, dann enthält ein Programm, das mit einer Menge von Testfällen getestet wurde, die diesem Kriterium genügen, keine Fehler mehr. Es ist nicht möglich für ein beliebiges Programm P ein gültiges Testauswahlkriterium zu bestimmen.
Angemessenheit: Ein Programm P ist angemessen zu einem gegeben Auswahlkriterium getestet, wenn die Menge der Testfälle, die diesem Kriterium genügt, keinen Fehler in P findet (d.h. erfolglos ist).

Axiome von Weyuker zur Angemessenheit von Tests:

- *Applicability:* Zu Programm P existiert eine angemessene Testfallmenge T
- *Non-Exhaustive Applicability:* Zu jedem Programm P existiert eine Testfallmenge T, die P angemessen testet und nicht alle denkbaren Testfälle enthält
- *Monotonicity:* Wenn die Testfallmenge T das Programm P angemessen testet, und T eine Untermenge von T’ ist, dann testet T’ P auch angemessen
- *Inadequate Empty Set:* Eine leere Testmenge testet kein Programm angemessen
- *Complexity:* Für jedes Programm P und ein Kriterium K gibt es eine Zahl n derart, daß P zwar durch eine K-konforme Testmenge der Kardinalität n, nicht aber durch eine K-konforme Testmenge der Kardinalität n-1 angemessen getestet werden kann.
- *Statement Coverage:* Wenn die Testfallmenge T das Programm P angemessen testet, dann wird dabei jede ausführbare Anweisung von P ausgeführt.
- *Renaming:* Werden Bezeichner umbenannt, dann testet eine vorher angemessene Testfallmenge auch die umbenannten Komponenten
- *Antiextensionalität:* Zwei semantisch identische Programme können nicht unbedingt auch mit derselben Testfallmenge getestet werden → Sie können unterschiedlich implementiert sein
- *Antidekomposition:* Wenn ein Programm P gemäß einem Kriterium angemessen getestet wurde, bedeutet dies nicht unbedingt, daß auch alle Komponenten Q angemessen getestet sind (Q kann Funktionen enthalten, die im Kontext von P nicht verwendet werden)
- *Antikomposition:* Wenn alle Komponenten eines Systems angemessen getestet wurden, bedeutet dies nicht unbedingt, daß auch das gesamte Programm angemessen getestet ist (Wenn alle Pfade in A und B durchlaufen werden, bedeutet dies nicht, daß auch alle Pfade durchlaufen werden, wenn ein Programm aus A und B zusammengesetzt wird).

Testfälle sollen repräsentativ, fehlersensitiv, redundanzarm und ökonomisch sein. Mit einer möglichst kleinen Auswahl der Testfälle soll möglichst vielen Fehlern auf die Spur gekommen werden.

Alternativen:

- Funktionaler Test (Black Box)
  - Ursache-Wirkungsgraphen
  - Eingabeüberdeckung
    - Äquivalenzklassenbildung
  - Ausgabeüberdeckung
    - Äquivalenzklassenbildung
  - Funktionsüberdeckung
  - Error Guessing
- Strukturorientierter Test (White Box)
  - Ablaufgraphenüberdeckung
    - Anweisungsüberdeckung
    - Zweigüberdeckung
    - Pfadüberdeckung
    - Termüberdeckung
  - Datenflußanalyse
  - Symbolischer Test
  - Diversifizierender Test

Die Abnahme ist eine besondere Form des Tests; Hier geht es nicht darum, Fehler zu finden, sondern zu zeigen, daß das System die gestellten Anforderungen erfüllt, d.h. in allen getesteten Fällen fehlerfrei arbeitet.

2.2.2. Black Box Test

Beim Black-Box Test ist der Ausgangspunkt für die Testfälle die Spezifikation. Fehlt diese, kann das Programm nur gegen sich selbst getestet werden, was keinen Nutzen bringt. Mit Hilfe dieser Testart soll eine möglichst umfassende Prüfung der Funktionalität durchgeführt werden können.
Von Nachteil ist die Tatsache, daß die konkrete Implementierung nicht geeignet berücksichtigt wird. Außerdem wird häufig mit einem reinen Black-Box Test nicht die Minimalanforderung eines White-Box Tests erzielt, die 100%ige Zweigüberdeckung.

Black-Box Testauswahlkriterien:

- *Funktionsüberdeckung:* Jede spezifizierte Funktion muß mindestens einmal ausgeführt werden
- *Eingabeüberdeckung:* Jedes Eingabedatum wir in mindestens einem Testfall verwendet (i. d. R. Probleme)
- *Ausgabeüberdeckung:* Jede Ausgabesituation wird mindestens einmal erzeugt (Beispiel: Bildschirmmasken, Fehlermeldungen, etc.)

Der Aufwand für diese Kriterien ist zum Teil erheblich (Beispiel: Textverarbeitung). Außerdem müssen häufig Funktionen in ihrem Zusammenspiel geprüft und Leistungs- und Robustheitsprüfungen gemacht werden.

Techniken der Testfall-Auswahl:

- *Äquivalenzklassenbildung:* Aus jeder Klasse von Eingabedaten werden Repräsentanten getestet
    - *Prinzip:* Zerlegung aller möglichen Eingaben in gültige und ungültige Äquivalenzklassen – Repräsentantentests. Aber: Äquivalenzklassen sind hypothetisch (Bildung nach Erfahrung und Intuition)
    - *Vorgehensweise:*
        - Aufstellung von Eingabebedingungen
        - Bildung von Äquivalenzklassen (Wertebereiche, Datenstrukturen, Aufzählungen)
        - Äquivalenzklassen identifizieren
        - Testfälle definieren für gültige Äquivalenzklassen (Auswahl, so daß möglichst viele gültige Ä-Klassen mit den Eingaben abgedeckt werden) → Wiederholung
        - Testfälle definieren für ungültige Äquivalenzklassen (Testfallauswahl derart, daß diese nur genau eine bisher noch nicht abgedeckte ungültige Ä-Klasse abdecken → sinnvoll: Zuordnung von Testfällen zu genau einer ungültigen Ä-Klasse → Klarheit über verursachende Eingaben)
    - *Diskussion:* Die Güte der Testfälle ist abhängig von der Aussagekraft der Spezifikation und die Definition der Testfälle hängt nicht nur von den Eingabebedingungen ab; Welche Werte einer Ä-Klasse sollen gewählt werden? Welche Kombinationen von Eingabebedingungen sollen getestet werden?
- *Grenzwertüberprüfung:* An Grenzen zulässiger Datenbereiche treten erfahrungsgemäß häufig Fehler auf
    - *Vorgehensweise:*
        - Testdaten identifizieren, die direkt auf oder neben den Grenzen der Ä-Klasse liegen (plus einen mittleren Wert)
        - Zusätzlich zu den Eingabeäquivalenzklassen werden Ausgabe-äquivalenzklassen für die erwarteten Resultate gebildet (manchmal nicht möglich, diese Eingaben zu finden, da ungültige Ausgaben nicht vom Programm zugelassen werden)
- *Ursache-Wirkungsgraphen:* Formale Sprache, in die eine Spezifikation übersetzt wird. Kombinationen von Eingabedaten, die zur Erzielung einer gewünschten Wirkung erforderlich sind, können bestimmt werden.
    - *Problem:* funktionale Äquivalenzklassen betrachten einzelne Eingaben, d.h. Wechselwirkungen und Abhängigkeiten werden nicht betrachtet. Die Anzahl der Kombinationsmöglichkeiten ist unter Umständen sehr hoch.
    - *Lösungsansatz:* systematische Auswahl von Eingabekombinationen
    - *Ursachen*-Wirkungsgraph ist das wesentliche Hilfsmittel, um zusätzliche Unvollständigkeiten und Widersprüche in der Spezifikation festzustellen
    - *Vorgehensweise:*
        - Zerlege die Spezifikation in einfach zu handhabende Teile (z.B. einzelne Kommandos)
        - Identifiziere Ursachen und Wirkungen
        - Erzeuge UWG durch semantische Transformation der Spezifikation
        - Forme den UWG zu einer Wertetabelle um
        - Für eine genaue Notation und eine Beispiel siehe Skript Nr. 06
- *Error Guessing:* Erfahrung zeigt, daß manche Mitarbeiter mehr Fehler finden als andere → mehr Erfahrung beim Testen, Intuition. Error Guessing ist kein systematisches Verfahren und ergänzt nur die Methoden zur Testfallbestimmung.

Diskussion des Black-Box Testens:
In der Praxis wird diese Art Test in den überwiegenden Fällen durchgeführt (bei System- oder Abnahmetests). Der Input besteht in der Regel aus der evtl. vorhandenen Spezifikation, dem Know-How der Entwickler sowie der Erfahrung und Intuition der Entwickler.
Methoden zur Testfallbestimmung werden nicht oder kaum verwendet, da sie nicht bekannt oder für große Systeme nur schwer anwendbar sind. Ein Beispiel sind UW-Graphen.
Bei Sicherheitskritischer Software werden Tests jedoch systematisch erarbeitet.

2.2.3. White Box Test

Bei dieser Art von Tests wird die Auswahl der Testfälle so vorgenommen, daß der Programmablauf oder der Datenfluß im Programm überprüft wird. Die Testfälle werden dabei so gewählt, daß das Programm systematisch durchlaufen wird.

Das zu testende Programm wird als Flußdiagramm betrachtet. *Anweisungen* werden durch *Knoten* und *Zweige* durch *Kanten nach Bedingungen* dargestellt. Als *Pfad* wird der Weg vom Anfangs- bis zum Endknoten bezeichnet. Sie werden durch alle Kombinationen aller Programmzweige bei maximalem Durchlauf aller Schleifen bestimmt.
Überdeckungen können nur rechnerunterstützt ermittelt werden. Testinstrumentierer fügen Zähler in Knoten und Kanten ein. Der Überdeckungsgrad wird über mehrere Läufe kumuliert.

Gebräuchlich sind drei Testüberdeckungen:

- *Anweisungsüberdeckung:*
  - *C0-Test:* Eine Testfallmenge T erfüllt das C0-Kriterium, wernn es für jede Anweisung A des Programms P einen Testfall gibt, der die Anweisung A ausführt. **Überdeckungsgrad: (ausgeführte Statements) / (Anzahl aller Statements)**. Black-Box – Testfälle produzieren ca. 60-70% Anweisungsüberdeckung, angestrebt werden 95-100%.
  - *Bewertung:* notwendiges, aber schwaches Kriterium, hilft Dead-Code zu finden, jedoch werden gewisse Kontrollflußfehler nicht immer gefunden.
- *Zweigüberdeckung:*
  - *C1-Test:* Eine Testfallmenge T erfüllt das C1-Kriterium, wenn es für jede Kante k im Kontrollflußgraph von P einen Weg in **Wege(t,P)** gibt, zu dem k gehört → d.h. wenn alle Entscheidungskanten ausgeführt werden. Black-Box – Testfälle produzieren ca. 80% Zweigüberdeckung, angestrebt werden 100%. ***Zweigüberdeckung wird in der Praxis angestrebt.***
  - *Bewertung:* minimales Testkriterium, hilft, dead code zu finden, berücksichtigt jedoch nicht komplexe Bedingungen.
- *Pfadüberdeckung:*
  C2-Test: intensivste kontrollflußorientierte Testmethode. Alle unterschiedlichen Pfade sollen einmal ausgeführt werden. Ein Pfad ist hierbei eine Sequenz von Knoten im Ablaufgraphen.
  - *Problem:* Pfadüberdeckung ist für reale Programme unrealistisch, da die Anzahl der Pfade durch Schleifen astronomisch hoch sein kann.
  - ***Eingeschränkte Pfadüberdeckung:*** Der C2-Überdeckungsgrad wird nur bei sicherheitskritischer Software angestrebt. Hier sollen nun nicht alle Pfade durch Schleifen berücksichtigt werden, sondern nur die Pfade, die neue Aspekte ansprechen.

        - ***Beschränkung auf 5 Aspekte pro Schleife:***
            - *0 Iterationen:* die Schleife wird nicht betreten
            - *1 Iteration:* zeigt häufig Initialisierungsfehler
            - *2 Iterationen:* Kann ebenfalls Initialisierungsfehler zeigen
            - *typ. Anzahl Iterationen:* Normalfall muß auch geprüft werden
            - *max. Iterationen:* Zeigt Fehler beim Abbruchkriterium
    - *Termüberdeckung:* Jeder atomare Term einer Bedingung bestimmt wenigstens einmal mit den werten TRUE und FALSE das Gesamtergebnis der Bedingung. (atomarer Term: Ein nicht mehr weiter zerlegbarer Teilausdruck mit booleschem Wert. Terme setzen sich rekursiv wieder aus Termen zusammen).

Vorteil dieses Testverfahrens: Formale Testauswahlkriterien können definiert werden.
Nachteil: Fehlende, in der Spezifikation beschriebene Funktionalitäten werden nicht erkannt.

2.3. objektorientiert

Seit einigen Jahren ist die Objektorientierung zum Standard – Entwicklungsansatz geworden. Sie gibt die Möglichkeit, die Wiederverwendung zu erhöhen. Dazu stellt sie Mechanismen wie Vererbung, Konstruktion nach dem open-closed-Prinzip und die Entwicklung von erweiterbaren Architekturen für Anwendungsfamilien bereit. Weiterhin werden potentiell wartbare Systemarchitekturen geschaffen. Dazu tragen die Prinzipien der Kapselung, der Kohäsion und Kopplung und der Vererbung bei. Industriestandards wie die UML oder JAVA wurden ebenfalls entwickelt.
Die Erfahrung zeigt jedoch, daß die hochgesteckten Erwartungen nicht erfüllt werden können, da auch OO-Systeme typische Qualitätsprobleme zeigen. Also müssen auch sie gewartet werden (OO Reengineering, Refactoring auf Code und Design-Ebene und Testverfahren für OO-Systeme).

Die Prinzipien und Ideen der bekannten Testverfahren (Unit-Test, Integrationstest, System, black / white Box-Test) bleiben unverändert, jedoch müssen sie in der Anwendung auf die Merkmale der Objektorientierung angepaßt werden. Was ist eine Unit in einem OO-System? Was mach man mit der Vererbung beim Testen?
→ Zum Teil sind neue Testverfahren notwendig (auch durch veränderte Vorgehensweise).

Eigenschaften von OO-Programmen:
- Es sind viele Klassen zu testen
- Klassen sind überschaubar, d.h. sie besitzen wenige Methoden und Attribute)
- Klassen müssen im Kontext der Vererbung betrachtet werden
- Klassen sind stark miteinander vernetzt
- Die logische Komplexität ist gering, die strukturelle hoch

→ Die grundlegenden Testprinzipien ändern sich nicht, jedoch ihre Granularität.

Einsatz traditioneller Verfahren:
- *Zweigüberdeckung:* wird schnell erfüllt, da die Methoden in der Regel klein sind und keine hohe Komplexität (Schachtelung von Kontrollstrukturen) haben. Sie ist sinnvoll einsetzbar bei komplexen Methoden, wenn nicht „gut“ objektorientiert programmiert wird. Sie sollte aber dennoch angestrebt werden, um den erzielten Überdeckungsgrad beim Test ermitteln zu können.
- *Black-Box-Verfahren:* Kaum Unterschied zu herkömmlichen Tests. Testfälle können aus objektorientierten Analysedokumenten ermittelt werden (z.B. Use Cases).

Außerdem kann die funktionelle Äquivalenzklassenbildung für den Test der Methoden- und Klassenschnittstellen verwendet werden.

Der Klassentest:
Die Klasse ist eine sinnvolle Einheit für den Test (Unit-Test).
Allerdings treten die folgenden Probleme auf:

- Klassen können nicht direkt getestet werden, da erst Objekte erzeugt und ggfs. Zunächst initialisiert werden müssen.
- Eine Klasse kann selbst Objektverhalten haben
- Nur die Schnittstellenoperationen können verwendet werden
- Die Kapselung verhindert die Sicht auf die interne Realisierung (Datenfelder, Hilfsoperationen)
- Die Aufrufreihenfolge der Methoden ist nicht immer festgelegt → Test sämtlicher Kombinationen
- Objekte verhalten sich wie Zustandsautomaten; Die Ergebnisse der Ausführung der Methoden können je nach Zustand unterschiedlich sein → Gleiche Methodensequenzen müssen mit unterschiedlichen Attributwerten getestet werden.

Einteilung von Klassen:

- *Nonmodal classes:* Es existiert *keine Einschränkung* für die Reihenfolge der Methodenaufrufe.
- *Unimodal calsses:* Methoden können nur in *festgelegten Reihenfolgen* aufgerufen werden.
- *Quasimodal classes:* Der *Zustand der Objekte* muß bei der Ausführung der Methoden berücksichtigt werden.
- *Modal classes:* Der *Zustand der Objekte und die Reihenfolgen* von Methodenaufrufen müssen berücksichtigt werden.

Richtlinien für den Klassentest:

- Jede Operation wird ausgeführt
- Alle Nachrichtenparameter und exportierten Attribute werden geprüft, indem Äquivalenzklassenvertreter und Grenzwerte verwendet werden
- Jede ausgehende Ausnahme wird angezeigt und jede eingehende behandelt
- Jeder Zustand wird erreicht
- Jede Operation wird in jedem Zustand ausgeführt (korrekt, wo erlaubt / verhindert, wo nicht erlaubt)
- Jeder Zustandsübergang wird erprobt um Zusicherungen zu testen
- Angemessene Belastungs- Leistungs- und Verdachtstest werden durchgeführt

Testen modularer Systeme:
Die Art der Vernetzung von Systemkomponenten beeinflußt die Testbarkeit von Systemen → Modulare Systeme sollen so entworfen werden, daß ihre Komponenten den *hohen internen Zusammenhalt* und *geringe Kopplung zu anderen Komponenten* aufweisen.
Die ***Kapselung*** erfüllt diese Kriterien.

Das Testen modularer Systeme hat den Vorteil, daß Module eine klare Schnittstelle nach außen besitzen und nur diese benutzt werden kann → Änderungen können auf der Ebene von Modulen durchgeführt werden, wenn die Schnittstelle stabil bleibt (nach außen hin ist nichts zu bemerken).
Ein Test nach einer Änderung eines Moduls (***Regressionstest***) würde intuitiv lauten, daß es genügt, das betreffende Modul nur noch einmal zu testen. Allerdings besagt das Axiom der

Antikomposition, daß auch alle vom geänderten Modul abhängigen Module getestet werden müssen → Daher sind Modul- und Integrationstest immer zusammen zu sehen. Da die Abhängigkeiten jedoch explizit durch Import-Listen gegeben sind, lassen die nachzutestenden Module gut bestimmen.

Testen objektorientierter Systeme:
Klassen können in der klassen *Benutzt-Beziehung*, aber auch in der *erbt-von-Beziehung* zueinander stehen, wobei die B*enutzt-Beziehung* nicht explizit im Code vermerkt ist. Durch Polymorphie erhält die *Benutzt-Beziehung* eine neue Bedeutung.

Die *Vererbung* führt dazu, daß flexible, offene Architekturen konstruiert werden können und lockert somit das Kapselungsprinzip, weil eine Klasse nicht mehr isoliert, sondern im Kontext ihrer Ober- und Unterklassen betrachtet werden muß. → *Dies gilt insbesondere für das Testen. Mehrfachvererbung und wiederholtes Erben führen zu Systemen, die schwer verständlich und auch fehleranfällig sind.*

Bei der *Modifikation* einer Klasse müssen alle auf dieser Klasse basierenden Klassen (*Benutzt-Beziehung*) und alle Unterklassen getestet werden → **Antikomposition**
Wird eine Klasse verändert oder eine neue Klasse in die Vererbungshierarchie eingefügt, so müssen alle Operationen der Oberklassen im Kontext der geänderten Klasse getestet werden (***Antidekomposition***). Ausnahme: Die eingefügte Klasse erweitert lediglich ihre Oberklasse.

Wird eine Methode m, die in der Klasse A definiert ist, in der Unterklasse B *redefiniert*, so muß diese im Kontext der Klasse B getestet werden. Hierzu wird im allgemeinen eine Menge von Testfälle benötigt, die unterschiedlich zu den Testfällen der nicht-redefinierten Methode in A ist (***Antiextensionalität***).

Der JOJO-Effekt:

Beim Jojo-Effekt kann es passieren, daß bei Aufrufen durch die Vererbungshierarchie gesprungen werden muß, da jede Oberklasse ihrerseits wieder Methoden der Oberklasse aufruft. Die Ursachen liegen beim Polymorphismus, der dazu führt, daß dynamisch zur Laufzeit die richtige Implementierung für eine Operation gefunden werden muß,“self“-Nachrichten., bei denen die Suche nach der richtigen Implementierung für eine solche Nachricht immer bei der Klasse des Empfängerobjekts beginnt und das rekursive Definieren, denn hierdurch können Operationen geschaffen werden, die selbst ihre Vorgängerimplementierung verwenden.
Durch das Redefinieren einer Operation in einer Klassenhierarchie kann sich die Wirkungs existierender Operationen verändern.
Beispiel zu finden in Vorlesung 11.

Auswirkungen auf das Testen:
Werden existierende Operationen in einer Klassenhierarchie redefiniert, so müssen unter Umständen Operationen der Oberklassen im Kontext der veränderten Klasse und Operationen der Unterklassen in ihrem Kontext erneut getestet werden.

Vollständiges Testen aller Klassen
Bei diesem Testverfahren werden alle Eigenschaften aller Klassen getestet, inklusive aller geerbten Eigenschaften. Dabei wird zuviel und teils unnütz getestet und die Möglichkeit verspielt, Testfälle und Test-Suites wiederzuverwenden.

Inkrementelles Testen
Ziel: Reduktion der Tests pro Klasse.
Die Idee dieses Verfahrens ist die Betrachtung aller Test-Suites als Eigenschaft einer Klasse. Unterklassen erben Test-Suites der Oberklassen(n).
Es soll dann automatisch berechnet werden, welche Eigenschaften der Unterklasse als neue Eigenschaft getestet werden müssen und welche der geerbten im Kontext der Unterklasse geänderten Eigenschaften getestet werden müssen.
Die Test-Suites müssen entsprechend modifiziert werden, d.h.:

- Welche Test-Suites der Oberklasse gelten auch für die Unterklasse?
- Welche Test-Suites der Oberklasse gelten nicht für die Unterklasse?
- Welche Test-Suites kommen neu hinzu?

Der ***Vorteil*** dieser Methode liegt darin, daß wirklich nur das getestet wird, was getestet werden muß und Test-Suites der Oberklasse werden, sofern möglich, wiederverwendet.

Vorgehensweise: Zuerst wird die Wurzelklasse einer Klassenhierarchie getestet. Für jede Eigenschaft der Klasse werden Test-Suites erstellt: Black-Box-Test-Suites ausgehend von der Spezifikation der Eigenschaft, White-Box-Test-Suites ausgehend von der Implementierung der Eigenschaft.
Somit erhält man ein inkrementelle Testen entlang der Vererbungsstruktur.

Tests in Unterklassen:
Für neue Methoden sind neue Testfälle zu entwerfen und auszuführen. Für redefinierte Methoden gilt dasselbe – zusätzlich müssen semantisch ähnliche Programm(teil)e mit unterschiedlichen Testfällen geprüft werden.
Für geerbte Methoden müssen alle Testfälle der Oberklasse erneut ausgeführt werden, da ihr Kontext in der Unterklasse ein anderer ist.

Zusammenfassend ist zu sagen, daß bei OO-Software neue Verfahren benötigt werden, die berücksichtigen, daß Klassen die atomaren Baueinheiten sind, die Vererbung die Kapselung auflockert und der Polymorphismus das Testen erschwert. Die Übernahme von herkömmlichen Methoden ist zudem nicht immer sinnvoll.

OO-Testmethoden sind noch Forschungsgegenstand. Solche Methoden werden im folgenden erläutert.

2.3.1. Zustandsbasierter Test

Das Thema sollte direkt aus dem Skript nachgearbeitet werden. Es eignet sich nicht für eine Zusammenfassung.

2.3.2. Use Case basierter Test

Bei diesem Test werden Use Cases (siehe OOSK) systematisch zur Testfallherleitung verwendet, indem diese in Zustandsdiagramme transformiert werden. Zusätzlich werden bei dieser Transformation die Use Cases zusätzlich und frühzeitig geprüft. Weiterhin werden die Probleme natürlichsprachlicher Spezifikation in den Use Cases durch diesen Formalisierungsschritt abgeschwächt. Allerdings ist die Transformation selbst nicht formalisiert, wird aber durch Richtlinien unterstützt. Eine automatische Generierung von Zustandsdiagrammen aus Use Cases existiert nicht → bei diesem zusätzlichen, manuellen, Modellierungsschritt können Fehler gemacht werden.
Die erstellten Zustandsdiagramme können nach Bedarf zu einem Gesamtsystem-Modell integriert werden.

Die ***SCENT-Methode*** stützt auf diese Transformation von Use Cases und reichert die Zustandsdiagramme mit Anmerkungen zur Testfallherleitung an: Vor- und Nachbedingungen, Datenbereiche, -Werte und erwartete Resultate, Leistungs- und nicht-funktionale Anforderungen).
Zunächst wird dann der Normalablauf modelliert, Alternativläufe kommen später hinzu. Ereignisse bilden die Zustandsübergänge → Einzelne Schritte in den Use Cases entsprechen Zuständen im Zustandsdiagramm.
Testfälle werden nun bestimmt, indem alle Pfade in den Zustandsdiagrammen durchschritten werden (Knoten-Kanten-Überdeckung oder beliebige andere Pfadüberdeckungsmaße).

Die Anmerkungen können genutzt werden, um mit Hilfe der Datenbereiche und Datenwerte Äquivalenzklassen zu bilden oder Grenzwertbetrachtungen durchzuführen. Zeit- und Leistungsanforderungen können ebenfalls aus ihnen ersichtlich sein.

2.3.3. Testautomatisierung

Die Motivation der Testautomatisierung liegt darin, daß Entwickler Code laufend verändern, Tests aber nicht bis zum vollständigen Abschluß der Entwicklung warten können. Beseitigung eines Fehlers führt häufig zu neuen und im Laufe der Programmwartung wird Code ebenfalls fortlaufend verändert.
Im Rahmen der Entwicklung objektorientierter Software ist die Entwicklung schnell und iterativ, separat entwickelte Klassen müssen integriert werden oder existierende Klassen werden in neuem Kontext wiederverwendet.

Ein *Regressionstest* ist ein erneuter Test des geänderten Programms mit bereits durchgeführten Tests. Dieser findet deutlich weniger Fehler als im ersten Durchlauf → geringere Effektivität. Automatisierung hilft hierbei, Kosten zu sparen. In der Praxis ist ein manueller Regressionstest eine Illusion.
Testaktivitäten im Bereich der Testimplementierung, der –Ausführung und dem Vergleich der Resultate sind gut automatisierbar. (meist wird Testautomatisierung als Ausführung und Ergebnisvergleich verstanden).

Testwerkzeuge haben keine Intelligenz, weswegen ein erfahrener Tester in bestimmten Situationen die bessere Wahl ist. Automatisierung ist nicht sinnvoll bei einmaliger Durchführung eines Tests, die Software sich häufig grundlegend ändert oder der Test leicht und / oder besser von Menschen durchgeführt werden kann (GUI-Test, etc.). Notwendige physische Interaktion ist ebenfalls eine Hemmschwelle für die Testautomatisierung.

Folgen der Automatisierung:
- Vergleich der erhaltenen und erwarteten Ergebnisse entscheidet für Testqualität
- Nur die Effizienz, nicht die Effektivität der Testfälle wird erhöht (es werden keine neuen Fehler gefunden, Testläufe werden nur schneller durchgeführt)
- Evtl. Einschränkung der Software-Entwicklung

Vorteile:
- Minimaler Aufwand für Test neuer Programmversionen
- Höhere Effizienz und höheres Vertrauen in die Software
- Erlaubt Tests, die nicht von Hand möglich oder zu aufwendig wären

- Bessere Nutzung der Ressourcen (keiner testet gerne, weniger fehleranfällig, Durchläufe nachts)
- Konsistenz und Wiederholbarkeit der Tests
- Bessere Testwiederverwendung (häufigere Verwendung von Tests, Testfallerstellung macht sich eher bezahlt)

Probleme:
- Automatisierung setzt eine bereits funktionierende, effektive Testpraxis voraus
- Effektivität nimmt ab, Vertrauen wird andererseits jedoch erhöht
- Trügerische Sicherheit
- Wartungsaufwand
- Technische Probleme (schlechte Qualität der Testwerkzeuge, Software oft schlecht testbar)
- Mangelnde Organisation (unrealistische Erwartungen, Notwendigkeit der Unterstützung seitens des Managements

Testgeschirr:
- Die Menge aller Dateien und Werkzeuge, die für den automatischen Test notwendig sind
- Implementierung der Testfälle / Testsuites z.B. als Skript
- Umgebung und Werkzeug für Testausführung
- Generierung von Teststümpfen, Messung der Testüberdeckung, Analyse

Voraussetzung für die erfolgreiche Automatisierung von Tests ist die modulare Strukturierung der Testfälle und Versionskontrolle. Weiterhin sollte die Vorgehensweise Unternehmensweit standardisiert sein. Zu testende Programme sollten komplett steuerbar sein → bei OO oft ein Problem

JUnit
Testrahmenwerk, um den Unit-Test eines JAVA-Programms zu automatisieren. Er ist einfach aufgebaut und leicht erlernbar → ein wichtiges Element des Extreme Programming
Die Entwickler programmieren Testfälle in JAVA – es sind keine speziellen Skriptsprachen notwendig. Idee: Inkrementeller Aufbau einer Testsuite parallel zur Entwicklung.
Eine *TestUnit* ist in erster Linie eine Klasse, hat jedoch keine klare Definition – alles, was sich als Test formulieren läßt, ist erlaubt.

2.4 Messen

Durch Messen werden *relevante Attribute* der betrachteten Entität *quantitativ* erfaßt. Meßergebnisse können *verglichen, bewertet* und *statistisch ausgewertet* werden.

Das Gebiet der Softwaremessung ist relativ neu und stützt sich auf die Leitideen, daß ohne Messung keine Kontrolle stattfinden und kein abstraktes Verständnis der Entwicklungen, die funktionieren oder nicht existieren, kann.

Ziele des Messens sind:
- Erfahrungen zu quantifizieren
- Entscheidungsgrundlagen zu gewinnen
- Prognosen zu stellen
- Qualität von Produkten und Prozessen lenken

Was soll jedoch bei Software gemessen werden?
Codeumfang ist kein geeignetes Kriterium.

Weitere Ausführungen siehe Vorlesung 15
Beispiele von Metriken und ihren Aussagen sind in Vorlesung 16 zu finden.

3. Konstruktive Maßnahmen (Prozeßmanagement)

3.1. Bewertung von SW-Prozessen - CMMI

CMMI (Capablility Maturity Model) ist eine standardisierte Möglichkeit, den Software-Entwicklungsprozeß zu bewerten und zu verbessern. Es kann im Sinne eines Self-Assessments genutzt werden, allerdings sind die Ergebnisse nur von Nutzen, wenn ehrlich geantwortet wird. Mit Hilfe dieser Bewertung können zentrale Schwächen im Softwareentwicklungsprozeß aufgezeigt werden. Somit eignet sich CMMI als gute Umgebungsbedingung, Verbesserungsmaßnahmen zu schaffen, erste solcher Maßnahmen zu planen und durchzuführen.

Es gibt insgesamt fünf Stufen dieses Modells, wobei jede Stufe auf die vorherige aufbaut. CMMI beschreibt lediglich, welche Fähigkeiten für die verschiedenen Reifegrade beherrscht werden müssen, schreibt aber keine spezifischen Methoden oder Werkzeuge vor.

Mit steigendem Reifegrad steigt die Vorhersagbarkeit von Terminen, Kosten, Qualität, die Produktivität der Projekte. Damit sinken das Risiko, daß Projekte ihr Ziel nicht erreichen und die Entwicklungszeit.

Stufen:

1.) **Performed:** Prozeß läuft ungeplant und chaotisch ab
2.) **Managed:** Software-Projektmanagement ist ausgebildet und institutionalisiert. Planung und Kontrolle der Projekte, systematische Erhebung von Anforderungen. Etablierung von Versions- und Konfigurationskontrolle, Fähigkeit der Produktmessung. Kontrollmechanismen für Kosten, Zeitplan, Budget und Qualität sind eingeführt → Qualitätssicherung ist etabliert
3.) **Defined:** Prozeß und alle Aktivitäten sind dokumentiert, standardisiert und eingebettet in unternehmensweites Prozeßmodell. Alle Projekte verwenden diesen Prozeß und passen ihn an.
4.) **Quantitatively Managed:** Detaillierte Maße (Metriken) für Produkte und Prozeß werden erhoben. Produkte und Prozesse können mit diesen Maßen überwacht werden.
5.) **Optimizing:** Permanente Prozeßverbesserung mit Hilfe der Maße. Einsatz neuer Methoden und Techniken, deren Wirkung überprüft werden kann.

Verteilung 2002: 2/3 der Unternehmen zu ähnlichen Teilen auf Stufen 1-3, ähnliche Verteilung des Rests auf Stufen 4 und 5.

Erfahrungen werden in Vorlesung 17 aufgeführt.

Der restliche Bereich der Vorlesung sollte direkt gelesen werden – eine Zusammenfassung ist nicht sinnvoll.