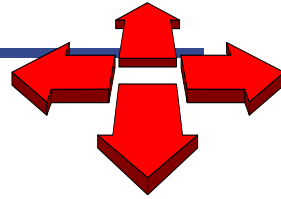




Heute auf der Agenda

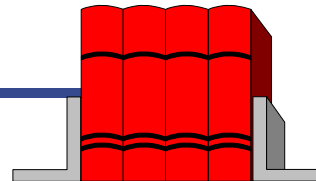
- Ein erneuter Blick auf das Thema **Testen**:
 - Qualität **analytisch** feststellen
- *5 Minuten Pause*
- Das **Vertragsmodell**:
 - Qualität **konstruktiv** erzeugen
- Zwei Ansagen vorab:
- Wir bieten wieder ein **Tutorium** an, in dem Inhalte aus Vorlesung und Übungen wiederholt und vertieft werden können.
 - **Termin: freitags 16 bis 18 Uhr in D-010**
 - Genauer Beginn (16:00, 16:15, 16:30) beim ersten Termin klärbar
 - **In der nächsten Woche Dienstag fallen beide Übungen aus!**
 - Alle Teilnehmer sollten sich alternative Termine suchen.

Testen Revisited



- Motivation
- Grundbegriffe (teilweise Wiederholung)
- Der Fehlerbegriff
- Grundsätze beim Testen
- Weitere Begriffe zum Thema Testen
- Probleme des objektorientierten Testens
- Testgetriebene Vorgehensweisen
- Weitere Testarten

Literaturhinweise



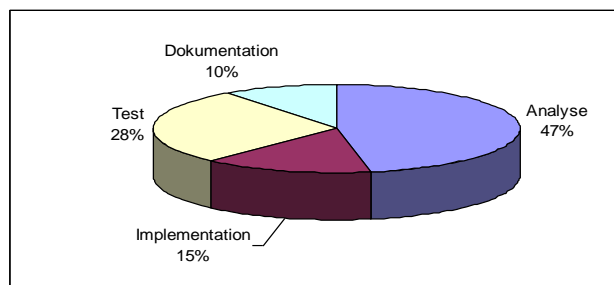
- Andreas [Spillner](#), Tilo [Linz](#), *Basiswissen Softwaretest*. 296 Seiten, Dpunkt Verlag 2005.
[DAS deutschsprachige Basisbuch zum Thema Testen; dient auch als Grundlage für den Certified Tester des ISTQB]
- Robert [Binder](#), *Testing Object Oriented Systems. Models, Patterns and Tools*. 1200 Seiten - Addison-Wesley Professional. November 1999.
[Standard-Buch über objektorientiertes Testen]
- Johannes [Link](#), Frank [Adler](#), Achim [Bangert](#), *Unit Tests mit Java. Der Test-First-Ansatz*. 348 Seiten - Dpunkt Verlag 2005.
[Gutes Buch über Test First mit JUnit]

Motivation für das Testen

- Wir haben bereits festgestellt:
 - „Ein fehlerfreies Softwaresystem gibt es derzeit nicht und wird es in naher Zukunft wahrscheinlich nicht geben, sobald das System einen gewissen Grad an Komplexität und Umfang an Programmzeilen umfasst.“^[Spillner, Linz]
- Dazu kommt:
 - Wir wollen (äußere und innere) Qualität eines Programms bestimmen.
 - Wir wollen die Funktionstüchtigkeit eines Programms erhöhen.
 - Wir wollen ein Programm besser verstehen, um es weiter zu entwickeln oder seine technische Qualität zu erhöhen.

Testen als Teil des „Programmverstehens“

- Laut Balzert kostet das Verstehen von Software bei der **Weiterentwicklung/Wartung** den größten Aufwand.
- Testen fördert das Verständnis von Software.



[Balzert98]

Aus SE1: Wann ist Software überhaupt „korrekt“?

- Die **Korrektheit** von Software kann immer nur in Relation zu ihrer **Spezifikation** gesehen werden - eine Software-Einheit ist korrekt, wenn sie ihre Spezifikation erfüllt.
- Der formale Nachweis, dass eine Software-Einheit ihre Spezifikation erfüllt, ist sehr aufwendig und schwierig.
- Voraussetzung für einen formalen Nachweis der Korrektheit ist, dass die Spezifikation selbst **formal definiert** ist. Dies ist nur sehr selten der Fall, meist sind Spezifikationen problembedingt nur informell formuliert.
- Auch wenn eine formale Spezifikation vorliegt: **Wie kann nachgewiesen werden, dass die Spezifikation selbst „korrekt“ ist?**

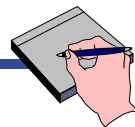
Ergo: Für umfangreiche interaktive Programme sind formale Korrektheitsbeweise heute nicht machbar.

Korrektheit ist in der Praxis somit häufig ein unscharfer Begriff.

Sehr viel nützlicher für die Praxis der Softwareentwicklung ist der Begriff der **Validität**.

Aus SE1: Statische und dynamische Tests

- Linz und Spillner unterscheiden in **Basiswissen Softwaretest** grundlegend zwischen statischen und dynamischen Tests.
- Ein **statischer Test** (häufig auch **statische Analyse** genannt) bezieht sich auf die Übersetzungszeit und analysiert primär den Quelltext. Statische Tests können **von Menschen** durchgeführt werden (Reviews u.ä.) oder mit Hilfe von **Werkzeugen**, wenn die zu testenden Dokumente einer formalen Struktur unterliegen (was bei Quelltext zutrifft).
- **Dynamische Tests** sind alle Tests, bei denen die zu testende Software ausgeführt wird.



Begriff: Statischer Test

- Statische Tests werden **nicht am laufenden System**, sondern an seinen Dokumenten (Quellcode und Dokumentation) ausgeführt.
- Sie sollen Fehler identifizieren und die Qualität verbessern.
- Statische Tests sind ein umfangreiches Gebiet. Hier werden nur die wesentlichen Verfahren benannt:
 - **Reviews:**
Prüfung der Dokumente durch Personengruppen nach festgelegten Regeln.
 - **Statische Analysen** (meist werkzeuggestützt):
 - Datenflussanalyse
 - Kontrollflussanalyse
 - Berechnung von Metriken

Wiederholung und Präzisierung: Was ist Testen?

- (Dynamisches) Testen von Software
 - Ein Testobjekt wird zur Überprüfung stichprobenartig ausgeführt.
 - Randbedingungen müssen festgelegt werden.
 - Die **Soll-Eigenschaften** werden anschließend mit den **Ist-Eigenschaften** verglichen.

erwartetes vs. geliefertes Verhalten

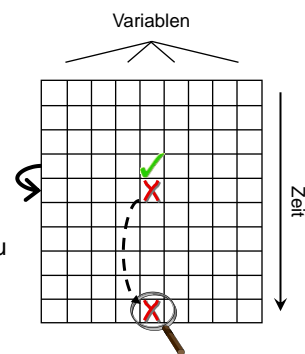
- Ziele des Testens:
 - **Fehlerwirkungen** nachweisen
 - Qualität bestimmen
 - Vertrauen und Verständnis schaffen
 - durch Analyse Fehlerwirkungen vorbeugen

Fehlerarten in Software (nach B. Meyer)

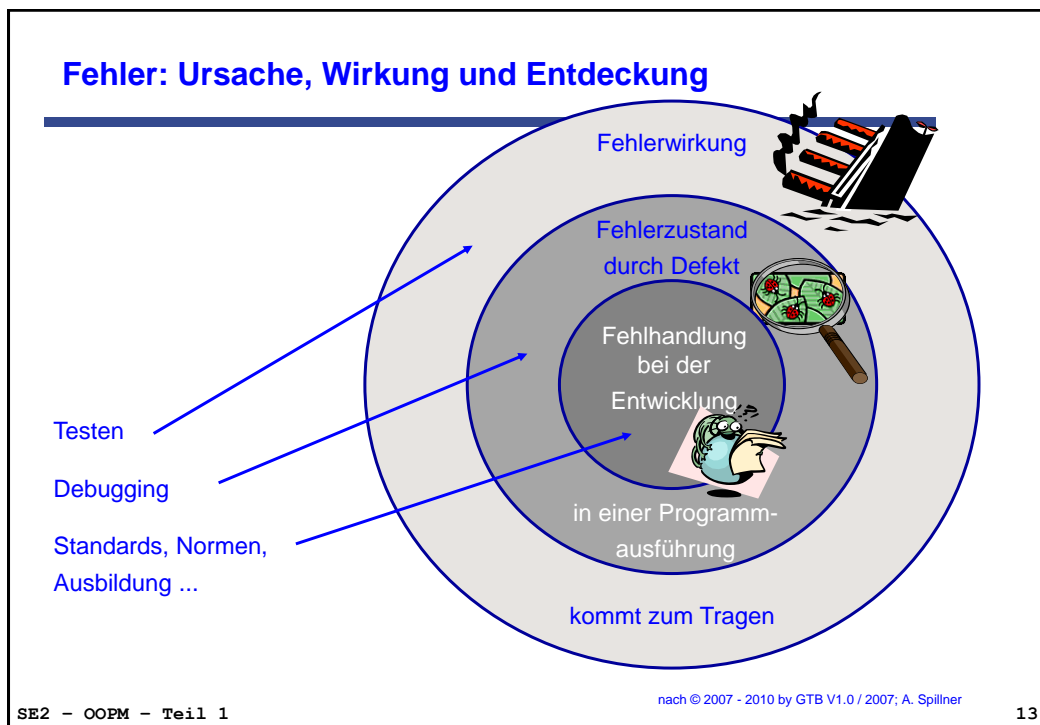
- Meyer differenziert Begriffe zu Fehlern in Softwaresystemen in folgender Weise:
 - Programmierfehler (engl.: error):**
Eine falsche Entscheidung, die während der Softwareentwicklung getroffen wurde.
 - Programmfehler (engl.: defect):**
Sie sind Folge von Programmierfehlern, „stecken“ in einer Software und **können** bei ihrer Ausführung bewirken, dass sich die Software nicht so verhält, wie dies gedacht war.
 - Laufzeitfehler (engl.: fault):**
Sie treten als Folge von Programm- oder Hardware-Fehlern zur Laufzeit auf. Ihr Effekt ist ein Programmabbruch, eine Fehlermeldung oder eine aus Sicht des Benutzers inakzeptable Systemreaktion.

Von Ursache zu Entdeckung: eine Motivation für das Vertragsmodell


- Zeller differenziert Laufzeitfehler **bei der Suche nach Fehlern** in Software etwas genauer:
 - defect** – der eigentliche **Defekt**, etwa eine fehlerhafte Anweisung in einer Quelltextzeile;
 - infection** – die **Verfälschung** des Speicherzustandes aufgrund eines Defektes, der zur Ausführung kommt;
 - failure** – den **extern beobachtbaren Fehler**, etwa durch einen Programmabbruch.
- Je **größer der Abstand** vom Wirken eines Defektes bis zu seiner Entdeckung, desto **schwieriger und damit teurer** das Finden der Fehlerursache.
- Das Vertragsmodell, konsequent umgesetzt, unterstützt beim schnelleren Aufdecken von Programmierfehlern!**



Zeller, A.: "Why Programs Fail – A Guide to Systematic Debugging", dpunkt-Verlag, 2006.



Grundsätze zum Testen (1)



In den letzten 40 Jahren haben sich folgende Grundsätze zum Testen herauskristallisiert und können somit als Leitlinien dienen:

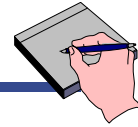
- **Testen garantiert nicht Fehlerfreiheit.**
Mit Testen wird die Anwesenheit von Fehlerwirkungen nachgewiesen. Testen kann nicht zeigen, dass keine Fehlerzustände im Testobjekt vorhanden sind!
»Program testing can be used to show the presence of bugs, but never to show their absence!« Edsger W. Dijkstra, 1970
- **Vollständiges Testen ist nicht möglich.**
Vollständiges Testen – Austesten – ist (abgesehen von wenigen Ausnahmen) nicht möglich.
(Beispiel kommt)

SE2 – OOPM – Teil 1

© Copyright 2007 - 2010 by GTB V1.0 / 2007; A. Spillner

14

Grundsätze zum Testen (2)



- **Mit dem Testen frühzeitig beginnen.**
Testen ist keine späte Phase in der Softwareentwicklung, es soll damit so früh wie möglich begonnen werden. Durch frühzeitiges Prüfen (z.B. Reviews) parallel zu den konstruktiven Tätigkeiten werden Fehler(zustände) früher erkannt und somit Kosten gesenkt.
- **Wo viele Fehler sind, verbergen sich meist noch mehr.**
Fehlerzustände sind in einem Testobjekt nicht gleichmäßig verteilt, vielmehr treten sie gehäuft auf. Dort wo viele Fehlerwirkungen nachgewiesen wurden, finden sich vermutlich auch noch weitere.

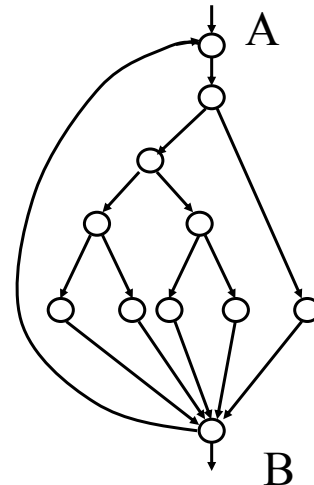
Grundsätze zum Testen (3)



- **Tests müssen gewartet werden.**
Tests nur zu wiederholen, bringt keine neuen Erkenntnisse. Testfälle sind zu prüfen, zu aktualisieren und zu modifizieren. Tests müssen also, genau wie Software, dynamisch Veränderungen angepasst werden, sonst sterben sie.
- **Testen ist abhängig vom Umfeld.**
Sicherheitskritische Systeme sind anders (intensiver, mit anderen Verfahren, ...) zu testen als beispielsweise der Internetauftritt einer Einrichtung.
- **Erfolgreiche Tests garantieren nicht Benutzbarkeit.**
Ein System ohne Fehlerwirkungen bedeutet nicht, dass das System auch den Vorstellungen der späteren Nutzer entspricht.

Austesten?

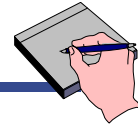
- Ein einfaches Programm soll getestet werden, das aus vier Verzweigungen (IF-Anweisungen) und einer umfassenden Schleife besteht und somit fünf mögliche Wege im Schleifenrumpf enthält.
- Unter der Annahme, dass die Verzweigungen voneinander unabhängig sind und bei einer Beschränkung der Schleifendurchläufe auf maximal 20, ergibt sich folgende Rechnung:
 $5^1 + 5^2 + \dots + 5^{18} + 5^{19} + 5^{20}$
- Wie lange dauert das Austesten bei 100.000 Tests pro Sekunde?
- Es sind 119.209.289.550.780 Testfälle
 Dauer: ca. 38 Jahre



Nochmals: Weshalb trotzdem Testen?

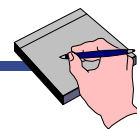
- Der **Quelltext kann leichter weiterentwickelt werden**, weil die Veränderungen einfach auf Korrektheit geprüft werden können.
- Die **Debugging-Zeiten reduzieren sich**, weil durch die Tests Fehler schneller lokalisiert werden können.
- **Schnittstellen werden einfach**, da jeder Programmierer lieber einfachere Schnittstellen testet. Dadurch wird auch vermieden, dass Technologie auf Vorrat gebaut wird.
- **Testklassen zeigen die vom Entwickler einer Klasse vorgesehene Verwendung** und können als ein Teil der Dokumentation des Quelltextes verstanden werden.
- Bei sog. „**Test First-Ansätzen**“ (kommt noch) kann das Testen als eine **Form der Spezifikation** der zu implementierenden Methode verstanden werden.

Begriffsvielfalt beim Testen



- Es gibt viele Arten des Testens und unterschiedliche Begriffe dazu. Wir unterscheiden grundlegend:
 - **Funktionale Tests:**
Das von außen sichtbare Ein-/Ausgabeverhalten des Testobjekts wird geprüft. Üblich sind dabei sog. Black-Box-Verfahren, mit denen die funktionalen Anforderungen an ein Programm geprüft werden.
 - **Nicht-funktionale Tests:**
Prüfen qualitativer Eigenschaften einer Software. Üblich sind Lasttest, Performanztest, Massentest, Stresstest, Test im Dauerbetrieb, Test auf Robustheit, Test auf Benutzerfreundlichkeit (Usability).
 - **Strukturbezogene Tests:**
Beziehen sich auf die interne Struktur und Architektur der Software (nach White-Box-Verfahren).
 - **Tests bezogen auf den Entwicklungsprozess:**
Den klassischen Aktivitäten im Entwicklungsprozess lassen sich Tests zuordnen: Komponententest, Integrationstest, Systemtest, Abnahmetest, Test nach Änderungen.

Aus SE1: Positives und negatives Testen



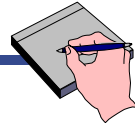
- Die gesamte Funktionalität einer Software-Einheit sollte durch eine Reihe von Testfällen überprüft werden.
- Ein **Testfall** besteht aus der Beschreibung der erwarteten Ausgabedaten für bestimmte Eingabedaten.
- Wenn nur **erwartete/gültige** Eingabewerte getestet werden, spricht man von **positivem Testen**.
- Wenn **unerwartete/ungültige** Eingabewerte getestet werden, spricht man von **negativem Testen**.
- Positive Tests erhöhen das Vertrauen in die **Korrektheit**, negative Tests das Vertrauen in die **Robustheit**.



Vorsicht:
Positiv/Negativ hat
hier nichts mit
true/false zu tun!



Aus SE1: Modultest und Integrationstest



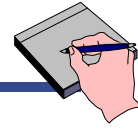
- Wenn die Einheiten eines Systems (Methoden, Klassen) isoliert getestet werden, spricht man von einem **Modultest** (engl.: unit test). Modultests sind eher technisch motiviert und orientieren sich an den programmiersprachlichen Einheiten eines Systems.
- Wenn alle getesteten Einzelteile eines Systems in ihrem Zusammenspiel getestet werden, spricht man von einem **Integrationstest** (engl.: integration test).
- Da erfolgreiche Modultests die Voraussetzung für Integrationstests sind, betrachten wir vorläufig nur Modultests näher.
- Die Methoden zum Modultest lassen sich grob in **Black-Box-**, **White-Box-** und **Schreibtischtests** unterteilen.
- Black-Box- und White-Box-Tests sind **dynamische Tests** (das Testobjekt wird ausgeführt), Schreibtischtests sind **statische Tests**.

Begriff: Komponententest



- Die kleinsten sinnvollen Bausteine im Entwicklungsprozess werden getestet.
 - **Unit-Test** oder **Modul-Test** sind eigentlich Begriffe aus der traditionellen imperativen Programmierung. Dort werden einzelne Prozeduren getestet.
 - Heute spricht man meist von **Komponententest**, der sich auf Methoden in Klassen, Klassen und ganze Subsysteme beziehen kann.
 - Durch **JUnit** (siehe SE1) hat der Begriff Unit-Test als automatisierter Regressionstest eine neue Interpretation bekommen.
- Geprüft werden die einzelnen Software-Bausteine isoliert von den anderen Systemteilen.
- Dabei soll sichergestellt werden, dass das Testobjekt die einzelnen funktionalen Anforderungen korrekt und vollständig realisiert.
- Teststrategien sind der **Whitebox-Test** oder **Test-first-Ansätze** (kommt noch).

Begriff: Test nach Änderungen / Regressionstest



- Das veränderte System wird getestet.
- Dabei soll sichergestellt werden, dass alle alten und ggf. die neuen Anforderungen vollständig realisiert sind.
- Teststrategien beim sog. Regressionstest sind:
 - Wiederholung aller Tests, die Fehlerwirkungen erzeugt haben
 - Vollständiger Test aller Programmkomponenten, die verändert worden sind
 - Test aller neuen Programmkomponenten
 - Vollständiger Test des gesamten Systems

Testen objektorientierter Software: nicht einfach!

- Die **objektorientierte Programmierung** besitzt gegenüber der klassischen imperativen Programmierung **strukturelle und dynamische Besonderheiten**, welche **beim Testen** zu berücksichtigen sind.
- Das Thema Testen von objektorientierten Programmen ist sehr umfangreich und wird in SE2 nur im Überblick behandelt.
- In den nächsten Vorlesungen werden wir wiederholt darauf zurückkommen, wenn grundlegende Konzepte eingeführt sind (beispielsweise Vererbung).
- Ausführliche Informationen sind in den Referenzen zu finden.

Probleme des oo Testens: Kapselung

- Der Mechanismus der **Kapselung** bedeutet, dass die **Implementation einer Operation verborgen** ist. Dieses softwaretechnisch wertvolle Prinzip erschwert den Test einer Klasse:
 - Reine Black-Box-Tests (siehe SE1) decken erfahrungsgemäß nur ein Drittel bis zur Hälfte der Zustände oder Ausführungspfade einer Klasse ab, da nur die nach außen sichtbare Struktur getestet werden kann.
 - Beim Testen ist es oft wichtig zu wissen,
 - welchen **konkreten Zustandsraum** ein Objekt haben kann,
 - wie die **Klasse strukturell eingebettet** ist,
 - welche **Abhängigkeiten** sich daraus ergeben.
- Dazu ist es notwendig, direkt auf den gekapselten Zustand eines Objektes zuzugreifen.

Probleme des oo Testens: Komplexe Abhängigkeiten

- Objektorientierte Software besteht aus Objekten. Objekte kommunizieren miteinander und ändern ihren Zustand durch Austausch von Nachrichten.
- Ob und wie ein Objekt auf eine Nachricht reagiert, definiert sich durch seinen eigenen Zustand und ggf. den Zustand, den es an einem anderen Objekt beobachten kann.
- Dadurch **bilden Objekte zur Laufzeit ein zeitabhängiges Netzwerk von kommunizierenden Einheiten** mit mehreren „Einstiegspunkten“ und ohne zentrale Instanz, die den Kontrollfluss steuert.
- Das macht das Testen von Kontrollflüssen sehr komplex:
 - Welches Netz von Objekten muss für einen Test aufgebaut werden?
 - In welchem Zustand müssen die Objekte sein?
 - Bei welchem Objekt beginnt der Test?

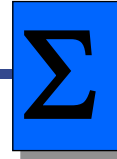
Testgetriebene Entwicklung

- **Agile Methoden** empfehlen, für jede Klasse eine eigene Testklasse zu schreiben.
- Beim sog. **Test First-Ansatz** sollen Testklassen **vor** den zu testenden Klassen geschrieben werden. Vor der Implementation der Methoden einer Klasse wird durch entsprechende Tests spezifiziert, welches Problem mit welchen Randbedingungen gelöst werden soll. So lässt sich auch eine gute Anweisungsüberdeckung erreichen.
- Testen und Programmieren sollen **in schnellem Wechsel** aufeinander folgen. Jede neue Klasse und Operation wird sofort getestet. Am Ende des Tages müssen alle Testfälle bei der Integration korrekt durchlaufen.
- Wir werden noch sehen: Insbesondere systematisches **Refactoring** erfordert Testklassen und automatisches Testen für eine sichere Veränderung in Einzelschritten.

Ausblick: Weitere Testarten

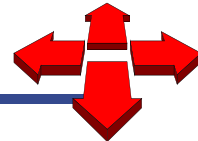
- **Akzeptanztests** sind Tests einer lauf- und einsatzfähigen Software-Version aus Sicht der Benutzung. Sie werden von Anwendern und Auftraggebern aus deren jeweiligen Blickwinkel (z.B.: Ist das System benutzbar? Erfüllt das System die vertraglichen Anforderungen?) durchgeführt.
- **Benchmark-Tests** messen die Performanz eines Systems gegen ein Vergleichssystem oder einen vorgegebenen Wert (z.B. Antwortzeit max. 0,5 Sekunden).
- **Lasttests** (Load Tests) prüfen das Verhalten des Systems bei praxisrelevanter Beanspruchung. Werden Extremsituationen (z.B. sehr große Benutzerzahlen oder hoher Datendurchsatz) getestet, spricht man von **Stresstests**.
- **Robustheitstests** prüfen, wie ein System auf Fehler, Ausnahmen oder nicht-spezifizierte Benutzereingaben reagiert.
- **Installationstests** prüfen, wie sich ein System in unterschiedlichen Installationskontexten verhält.

Zusammenfassung Testen



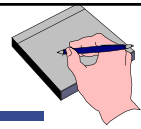
- Testen in der Softwareentwicklung dient u.a. dem **Aufdecken** von Fehlern durch den Abgleich von **Soll-** und **Ist-Werten**.
- **Fehler** können differenziert werden in
 - **Fehlhandlungen** (Programmfehler, engl.: error) von Personen, die zu Defekten führen;
 - **Fehlerzustände**: statisch in Programmtexten (Programmfehler oder Defekt, engl.: defect) oder im Speicher bei der Ausführung (Laufzeitfehler, Verfälschung, engl.: fault, infection);
 - **Fehlerwirkungen**: eine Abweichung zwischen Soll und Ist wird entdeckt, Programmabbruch (engl.: failure).
- Zum Thema Testen gibt es eine große **Begriffsvielfalt**.
- Das Testen von **objektorientierter Software** hat einige Besonderheiten, die wir in den nächsten Wochen näher untersuchen wollen.

Das Vertragsmodell



- Aus SE1 kennen wir (generische) Java-Datentypen wie **Stack**, **Queue**, **List** und **Set**.
- Das **Verhalten** solcher Datentypen kann programmiersprachen-unabhängig spezifiziert werden, in Form so genannter **Abstrakter Datentypen** (engl.: abstract data types, häufig abgekürzt mit **ADT**).
- Bevor wir uns ADTs näher ansehen, wollen wir zuerst eine pragmatische Anwendung der theoretischen Konzepte abstrakter Datentypen betrachten: das **Vertragsmodell** der objektorientierten Softwareentwicklung.
- Das Vertragsmodell ermöglicht unter anderem, das Grundkonzept der Objektorientierung, das Verhältnis zwischen **Klient** und **Dienstleister**, klarer zu fassen.

Vertrag

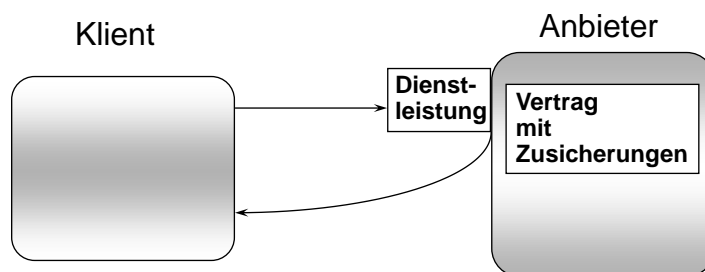


Nach Wikipedia:

Ein **Vertrag** ist eine von zwei oder mehreren Vertragspartnern geschlossene Übereinkunft. Er dient der Herbeiführung eines von den Parteien gewollten Erfolges. Der Vertrag kommt durch übereinstimmende Willenserklärungen zustande.

Im Vertragsmodell:

Ein **Klient** und ein **Anbieter** schließen einen **Vertrag** über eine Dienstleistung. Die Bedingungen des Vertrags sind als wechselseitige Zusicherungen formuliert und werden vom Anbieter verwahrt.



SE2 – OOPM – Teil 1

31

Das Vertragsmodell der objektorientierten SW-Entwicklung

Die Metapher:

- Die Benutzt-Beziehung zwischen Klassen wird als Vertragsverhältnis zwischen **Klient** (engl.: Client) und **Lieferant** (engl.: Supplier) interpretiert:
- Eine Klasse als Lieferant bietet eine Dienstleistung an, die eine andere Klasse als Klient nutzen will.
- Im Vertrag wird formuliert,
 - welche Vorleistung der Klient erbringen muss,
 - damit der Lieferant seine Dienstleistung liefert
 - und dies auch garantiert.
- Der Begriff **Vertragsmodell** ist unsere deutsche Übersetzung des englischen **Design by Contract**, wie es von Meyer geprägt wurde.



SE2 – OOPM – Teil 1

32

Benutzt-Beziehung und Vertragsmodell

- Eine **Benutzt-Beziehung** zwischen zwei Klassen **A** und **B** wird (in Java) auf zwei Arten hergestellt:
 - In Klasse **A** werden Variablen vom Typ **B** deklariert.
 - In einer Methode der Klasse **A** werden Operationen an Exemplaren der Klasse **B** aufgerufen.
- Das **Vertragsmodell** bezieht sich auf:
 - den Aufruf von Operationen,
 - die Überprüfung von Aufrufparametern sowie des Zustands des gerufenen Exemplars.

Die Bestandteile des Vertragsmodells

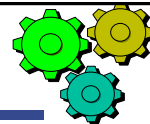
- **Der Vertrag:**
 - Ein Vertrag bezieht sich immer auf eine Operation einer Klasse.
 - Er wird in der Lieferanten-Klasse festgelegt.
 - Die Vertragsbedingungen werden als Zusicherungen spezifiziert.
- **Zusicherungen:**
 - Zusicherungen sind **boolesche Ausdrücke** (Prädikate). Es gibt:
 - **Vorbedingungen** (vor der Ausführung der Operation)
 - **Nachbedingungen** (nach der Ausführung der Operation)
- Zusätzlich gibt es **Invarianten**:
 - Bedingungen, die im Vertragsmodell immer gelten sollen, werden als **Klassen-Invarianten** festgehalten, die ebenfalls **boolesche Ausdrücke** sind.

Der Mechanismus des Vertragsmodells

- Ein Vertrag wird bei jedem Operationsaufruf geprüft:
 - Der Klient muss sicherstellen, dass die Vorbedingungen der Operation erfüllt sind. Beim Lieferant muss geprüft werden, ob die Vorbedingung gilt.
 - Wenn die Vorbedingung erfüllt ist, führt der Lieferant die Operation aus.
 - Der Lieferant garantiert dann durch die Nachbedingungen, dass die Leistung erbracht ist.
- Die Klasseninvariante ist eine allgemeine Randbedingung des Vertrags. Sie muss bei jedem Operationsaufruf gelten.

Merke:
Wenn
 der **Klient** die **Vorbedingungen** erfüllt,
dann
 garantiert der **Lieferant** die **Nachbedingungen**.

Ein Beispiel aus einer Bibliothek



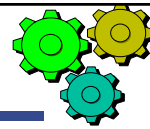
Die fachliche Schnittstelle der Klasse **Buch** ist aus den Aufgaben und Tätigkeiten von BibliothekarInnen und BibliotheksbenutzerInnen abgeleitet.

Buch

ausleihen (b : Benutzer)
 zurückgeben (b : Benutzer)
 mahnen (b : Benutzer)
 verlängern (b : Benutzer, f : Frist)
 vorbestellen (b : Benutzer)

- Probleme bei der Verwendung
 - Wir haben ein Vorverständnis von der Bedeutung der Operationen an der Schnittstelle, kennen aber die genaue Semantik nicht.
 - Vermutlich lässt sich nicht jede Operation an einem Buch-Objekt jederzeit aufrufen.
 - Es gelten vermutlich Regeln für die Verwendung der Operationen ("ein Buch muss zurückgegeben sein, ehe es ausgeliehen werden kann").

Auf dem Weg zum Vertragsmodell (1)



Wir formulieren Vor- und Nachbedingungen aus fachlicher Sicht.

Vorbedingungen werden durch **require** gekennzeichnet, Nachbedingungen durch **ensure** (Der Syntax der Sprache Eiffel folgend).

Buch

```
ausleihen (b : Benutzer)
    require: NOT (ist_ausgeliehen())
    ensure: ist_ausgeliehen()

zurückgeben (b : Benutzer)
    require: ist_ausgeliehen()
    ensure: NOT (ist_ausgeliehen())

...
```

Rechte und Pflichten des Klienten

- Der Klient hat die **Pflicht**, die Operation nur aufzurufen, wenn der **Zustand des Lieferanten** es erlaubt.
- Dann hat der Klient aber auch das **Recht**, die **ordnungsgemäße Erfüllung** des Vertrags zu erwarten.



Einhalten der Vorbedingungen



Erfüllung der Nachbedingungen

- Das Vertragsmodell wurde bisher nur in der Sprache Eiffel direkt umgesetzt.



```
public class Buch                                Lieferant
{
    //Vorbedingung: NOT (ist_ausgeliehen())
    public void ausleihen(Benutzer b)
    {
        ...
    }
    //Nachbedingung: ist_ausgeliehen()

    public boolean ist_ausgeliehen()
    {
        ...
    }

    private boolean _ausgeliehen;
    private Benutzer _b;
}
```

Rechte und Pflichten des Lieferanten

- Der Lieferant hat die **Pflicht**, die **versprochene Leistung** zu erbringen.
- Er hat aber gleichzeitig das **Recht** bei Nichteinhaltung des Vertrags von Seiten des Klienten, die **Operation nicht auszuführen**.



Erfüllung der
Nachbedingungen



Einhalten der
Vorbedingungen

Klient

```
Benutzer einBenutzer = new Benutzer("hz");
Buch einBuch = new Buch();

einBuch.ausleihen (einBenutzer); ✓

einBuch.ausleihen (einBenutzer); ✗
```

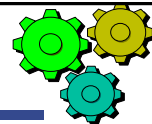
Lieferant

```
public class Buch
{
    //Vorbedingung: NOT (ist_ausgeliehen())
    public void ausleihen(Benutzer b)
    {...}
    //Nachbedingung: ist_ausgeliehen()

    public boolean ist_ausgeliehen()
    {...}

    private boolean _ausgeliehen;
    private Benutzer _b;
};
```

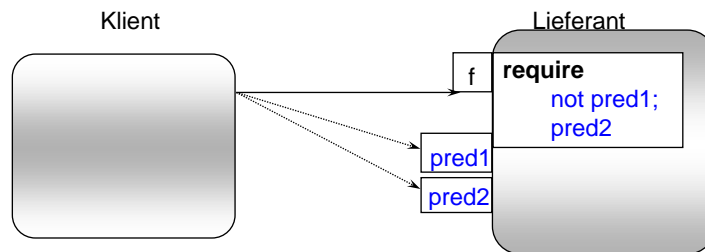
Auf dem Weg zum Vertragsmodell (2)



Damit der Klient seine Verpflichtungen erfüllen kann, muss er die entsprechenden Vorbedingungen des Lieferanten prüfen können. Dazu muss die Schnittstelle des Lieferanten entsprechende sondierende Operationen enthalten. Dabei wird der Zusammenhang zwischen den Operationen `ausleihen()` und `zurückgeben()` deutlich.

Buch
ausleihen (b : Benutzer) require: ist_ausleihbar() ensure: ist_ausgeliehen()
zurückgeben (b : Benutzer) require: ist_ausgeliehen() ensure: ist_ausleihbar()
ist_ausleihbar () : boolean ist_ausgeliehen () : boolean

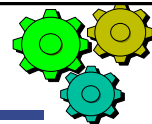
Der Klient muss seine Verpflichtungen prüfen können



Vertragsmodell und Zusicherungen:

- Damit ein Vertrag sinnvoll ist, muss der Klient die Einhaltung der Vorbedingung einer zu rufenden Operation prüfen können.
- Daher müssen alle in einer Vorbedingung verwendeten Terme geeignet als Prädikate (sondierende Operationen) an der Schnittstelle sichtbar sein.
- Da die Einhaltung der Nachbedingung Aufgabe des Lieferanten ist, müssen die entsprechenden Terme nicht Teil der Schnittstelle werden.

Auf dem Weg zum Vertragsmodell (3)



Alle in Vorbedingungen verwendeten Terme werden geeignet in Prädikate (sondierende boolesche Operationen) umgesetzt.

- Prädikate sind total, d.h. sie können in jedem Zustand des Objekts gerufen werden; sie dürfen also selbst keine Vorbedingungen haben.
- Prädikate haben keinen von außen sichtbaren Effekt auf den Zustand des Objekts.

Buch
ist_ausleihbar() : boolean
ist_ausgeliehen() : boolean



- Die beiden Prädikate `ist_ausleihbar()` und `ist_ausgeliehen()` sind immer an Exemplaren der Klasse `Buch` ausführbar. Bei ihrer Implementation ist darauf zu achten, dass keine Zustandsvariablen durch Zuweisung verändert werden.

Operationen an der Schnittstelle

Die Schnittstelle einer Klasse sollte den Prinzipien eines ADT entsprechend so aufgebaut sein:

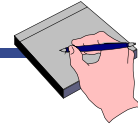
- Die **verändernden Operationen**:
Verändernde Operationen haben Seiteneffekte, d.h. verändern den Zustand eines Objekts: sie haben selten einen Rückgabewert (z.B. **pop** verändert den Stack; liefert aber kein Element).
- Die **sondierenden fachlichen Operationen**:
Sie liefern einen fachlichen Rückgabewert (Fachwert) oder ein Rückgabeobjekt.
(z.B. **top** liefert ein Stack-Element).
Eine sondierende Operation hat keine Seiteneffekte, d.h. der (von außen sichtbare) Zustand eines Objektes bleibt unverändert.
- Die **sondierenden booleschen Operationen**:
Sie prüfen den jeweiligen Objektzustand und werden besonders zur Sicherung von Vorbedingungen verwendet.

Zustandsabhängigkeit von Operationen

Nicht jede Operation eines Objekts ist zu jedem Zeitpunkt aufrufbar:

- Die **verändernden Operationen**:
Sie sind selten zu jedem Zeitpunkt im Lebenszyklus eines Objekts aufrufbar. Dies sollten die Vorbedingungen deutlich machen. Da ihr Aufruf den Objektzustand verändert, muss **vor jedem Aufruf** erneut die Zulässigkeit geprüft werden.
- Die **sondierenden fachlichen Operationen**:
Sie sind ebenfalls meist nur an bestimmten Zeitpunkten im Lebenszyklus eines Objekts aufrufbar. Dies sollten die Vorbedingungen deutlich machen. Durch ihren wiederholten Aufruf sollte sich nichts an der Zulässigkeit verändern.
- Die **sondierenden booleschen Operationen**:
Sie sind zu jedem Zeitpunkt aufrufbar. Sie dürfen keine Vorbedingungen haben.

Klassen-Invarianten



Klassen-Invariante (meist kurz Invariante):

- Eine Klassen-Invariante gilt für alle Exemplare einer Klasse und muss von allen Operationen berücksichtigt werden. Sie beschreibt (semantische) Randbedingungen einer Klasse insgesamt.
- Formal ist sie eine boolesche Aussage über alle Exemplare einer Klasse, die *vor* und *nach* Ausführung jeder Operation der Klasse gelten muss.

- Beispiel 1:
Wir modellieren zunächst, dass ein Buch ausgeliehen oder ausleihbar ist:
`Invariant: ist_ausgeliehen() ^ ist_ausleihbar()`
- Beispiel 2:
Wenn wir berücksichtigen, dass ein Buch vorbestellt werden kann, dann erweitert sich die Invariante zu:
`Invariant: ist_ausgeliehen() ^ ist_ausleihbar();
NOT ist_ausleihbar() ^
 (ist_vorhanden() & NOT ist_vorbestellt())`

Korrektheit einer Klasse

Korrektheit bezogen auf die Zusicherungen:

- Die Korrektheit eines Objektes kann nur in sog. **stabilen Zuständen** geprüft werden, d.h. vor und nach der Ausführung von Operationen.
- Dann muss auch jeweils die Klasseninvariante gelten.
- Zwischenzeitlich kann die Invariante (z.B. während des Aufrufs einer privaten Methode) verletzt sein.

- Eine Klasse K ist im Sinne des Vertragsmodells korrekt, wenn:
 - **beim Erzeugen eines Objekts** die Vorbedingung des gerufenen Konstruktors erfüllt ist, und die Nachbedingung und die Invariante vom gerufenen Konstruktor erfüllt wird.
 - **für jede gerufene Operation** die Vorbedingung und die Invariante gelten, und die Nachbedingung und die Invariante von der gerufenen Operation erfüllt wird.



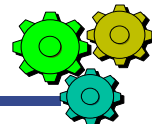
Kein eingebautes Vertragsmodell in Java

- Java bietet **keine Sprachunterstützung** für das Vertragsmodell; d.h. Vor- und Nachbedingungen und Invarianten sind **nicht Teil des Sprachmodells**.
- **James Gosling**, der Designer von Java, hat in einem Interview gesagt, dass er anfangs das Vertragsmodell in die Sprache integrieren wollte. Das mangelnde Verständnis der meisten Programmierer der damaligen Zeit für das Konzept hat ihn dann aber davon abgehalten, was er inzwischen bedauert.
- Wenn wir das Vertragsmodell in Java umsetzen wollen, müssen wir es deshalb **„von Hand“ (manuell) programmieren**.



„The C Family of Languages: Interview with Dennis Ritchie, Bjarne Stroustrup, and James Gosling“, Java Report, 5(7), July 2000.

Manuelle Umsetzung des Vertragsmodells in Java

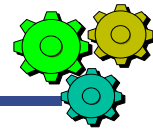


- Wenn wir das Vertragsmodell in Java umsetzen wollen, dann müssen wir sowohl die **Dokumentation** der Verträge als auch deren **Überprüfung** bedenken.
 - Die **Dokumentation** sollte im **Schnittstellenkommentar** einer Operation stehen, damit die Vertragsinformationen für einen Klienten ersichtlich sind.
 - Die **Überprüfung** kann nur innerhalb des Methodenrumpfes erfolgen, der die Operation implementiert:
 - Vorbedingungen sollten unmittelbar zu Beginn geprüft werden.
 - Nachbedingungen sollten beim Methodenausgang geprüft werden; bei einer **return**-Anweisung kann die Prüfung aber nur unmittelbar vor dieser Anweisung erfolgen.
 - Die Invarianten werden meist nur im Klassenkommentar notiert.



- Die Trennung von Dokumentation und Überprüfung führt zu einem potenziellen Wartungsproblem: Bei Änderungen kann es leicht zu Inkonsistenzen zwischen Programmtext und Kommentar kommen!

Überprüfung von Verträgen in Java



- Für die manuelle Überprüfung gibt es verschiedene Möglichkeiten:
 - Eine **spezifische Fehlerbehandlung für jeden Einzelfall** mit jeweils passendem Exception-Typ.
 - Eine zentrale Implementation, z.B. über Klassenmethoden einer Contract-Klasse wie im **Framework JWAM**:



```
public static void require (boolean precondition,
                           Object contractor, String description);
```

Vorteil: Knappere Darstellung, einheitliche Behandlung

Nachteil: Die Aufrufe lassen sich bei Bedarf nicht leicht entfernen.

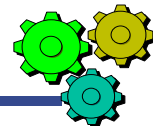
- Verwendung der **assert-Anweisung** von Java.

Vorteil: ebenfalls knapper, zusätzlich (zur Laufzeit) abschaltbar

Nachteil: In Java bedeutet abschaltbar leider, dass das Anschalten vergessen werden kann (die Überprüfung von Assertions ist standardmäßig **nicht** angeschaltet).



Vertragsprüfung in Java mit assert



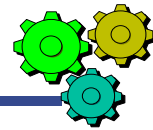
- Die **assert-Anweisung** von Java gibt es in zwei Varianten:


```
assert BooleanExpression ;
assert BooleanExpression : Expression ;
```
- Die zweite Variante kann für das Vertragsmodell verwendet werden, indem der boolesche Ausdruck die eigentliche Zusicherung formuliert, während der Ausdruck nach dem Doppelpunkt einen String mit der Beschreibung der Zusicherung liefert.
- Beispiel für unsere altbekannte Konto-Klasse:

```
/**
 * Zahle einen Betrag auf dieses Konto ein.
 * @require betrag >= 0
 */
public void einzahlen (int betrag)
{
    assert betrag >= 0 : "Vorbedingung verletzt: betrag >= 0";
    ...
}
```



Diskussion: `assert` für Vertragsmodell in Java



- Sun Microsystems rät explizit von der Verwendung der `assert`-Anweisung für Vorbedingungen ab, da die Überprüfung von Asserts eine Laufzeit-Option ist, die ausgeschaltet werden kann.
- Diese Position ist aus der **Sicht eines API-Anbieters** formuliert; Bibliothekscode sollte bei der Überprüfung seiner Vorbedingungen nicht davon abhängig sein, ob zufällig auf der ausführenden Virtual Machine die Assertions geprüft werden. Stattdessen sollten die Vorbedingungen explizit überprüft werden und bei einer Verletzung eine passende Runtime-Exception ausgelöst werden (etwa eine `InvalidArgumentException`).
- Wir teilen dieses Argument nicht ganz, denn zumindest **innerhalb eines Software-Entwicklungsprojektes** besteht die Kontrolle über die ausführende Virtual Machine.
- Weiterhin ist das Vertragsmodells primär eine Hilfe bei der Entwicklung; in einem ausgelieferten System müssen die Verträge nicht mehr zwingend überprüft werden.
- Aber auch in einem laufenden System können durch die Prüfung Inkonsistenzen früher erkannt werden...



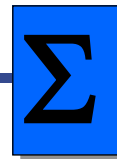
Vertragsmodell und Ausnahmen

- **Verträge müssen erfüllt werden** oder es gibt eine **Vertragsverletzung**; eine Vertragsverletzung aber ist ein **Programmierfehler**!
- Folglich sollten fehlgeschlagene Zusicherungen zu einem **Programmabbruch** führen, bevor Schlimmeres passieren kann.
- Moderne Sprachen bieten dazu das Konzept von **Ausnahmen** (engl.: exceptions), das wir (zumindest für Programmabbrüche) bereits in SE1 kennen gelernt haben (mehr zu Ausnahmen/Exceptions später).
- Nach dem Vertragsmodell tritt eine **Ausnahmesituation** auf, wenn
 - der Klient die Vorbedingungen nicht liefert (Fehlerursache liegt beim Klienten), oder
 - der Lieferant die Nachbedingungen und Invarianten nicht erfüllen kann (Fehlerursache liegt beim Lieferanten).
- **Grundsätzlich:**
Ein Vertrag wird erfüllt oder die Vertragsverletzung wird festgestellt, d.h. Operationen sind erfolgreich oder schlagen mit einer Ausnahme fehl.

Vertragsmodell und Testen: Vorsicht bei Negativtests!

- Durch das Vertragsmodell können die **Zuständigkeiten** von **Klient** und **Dienstleister** klarer getrennt und definiert werden.
- Das Vertragsmodell liefert **Hinweise auf Testfälle**: Wenn ein Klient sich an die Vorbedingungen hält, kann er die Nachbedingungen erwarten – und sie in geeigneten Testfällen überprüfen.
- Eine Merkregel aus SE1 lautete: Positivtests müssen, Negativtests können sein! Anders gesagt: **Korrektheit muss, Robustheit kann** geprüft werden.
- Im Zusammenhang mit dem Vertragsmodell gilt: **Negativtests zum Vertragsmodell** (also bewusstes Verletzen der Vorbedingungen, um zu überprüfen, ob der Dienstleister sie überprüft) sind zu **vermeiden**.
- Warum das?
- Weil die **assert-Prüfungen gefahrlos ausschaltbar** sein sollten. Dies wäre bei Negativtests des Vertragsmodells jedoch nicht der Fall, da die Testfälle fehlschlagen würden, die erwarten, dass bei einer verletzten Vorbedingung beispielsweise eine bestimmte Exception geworfen wird.

Zwischenergebnis Vertragsmodell



- Das **Vertragsmodell** ist eine Interpretation der **Benutzt-Beziehung zwischen Klassen**.
- Es unterstützt
 - bei der Formulierung von Konsistenzbedingungen über Klassen und Objekte;
 - die Entwicklung verständlicher und fachlich „runder“ Klassen;
 - eine disziplinierte Fehlerbehandlung.
- **Zusicherungen** (Vor- und Nachbedingungen) und **Randbedingungen** (Invarianten) sind beim Dienstleister (der Angebote macht) formuliert.
- Vor- und Nachbedingungen und Invarianten sollen **keine Repräsentationsdetails**, sondern „vertragsrelevante“ Daten enthalten.
- Wir werden das Vertragsmodell noch einmal im Zusammenhang mit **Vererbung** diskutieren.