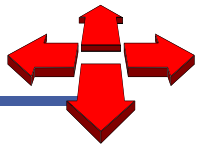
Teil 1: Abstraktion, Vertragsmodell, Fehlerbehandlung, Polymorphie

Motivation des Vertragsmodells

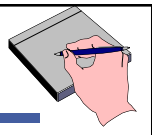
- **Wiederverwendbarkeit** hängt ab von:
  - **Verständlichkeit**,
    (Sinn und Zweck einer Klasse müssen verstanden werden),
  - sicherer **Verwendbarkeit**,
    (die Klasse muß als korrekt und robust eingeschätzt werden).
- Das "reine" objektorientierte Modell liefert:
  - den Klassennamen,
  - die Signaturen von Routinen.
- Es fehlt:
  - Spezifikation der Semantik

Das Vertragsmodell



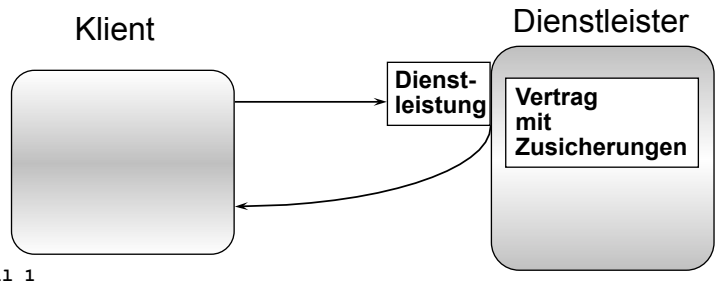
- Aus SE1 kennen wir (generische) Java-Datentypen wie `Stack`, `Queue`, `List` und `Set`.
- Das **Verhalten** solcher Datentypen kann programmiersprachen-unabhängig spezifiziert werden, in Form so genannter **Abstrakter Datentypen** (engl.: abstract data types, häufig abgekürzt mit **ADT**).
  - Beispiel: *top(push(s, x)) = x*
  - Lies: „Wenn ich ein Element *x* auf einen Stack *s* lege, dann erhalte ich einen neuen Stack, dessen oberstes Element *x* ist.“
- 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 u.a., eines der Grundkonzepte der Objektorientierung, nämlich 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 **Dienstleister** schließen einen **Vertrag** über eine Dienstleistung. Die Bedingungen des Vertrags sind als wechselseitige Zusicherungen formuliert und werden vom Dienstleister verwahrt.



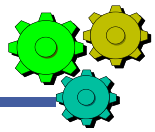
Das Vertragsmodell der objektorientierten SW-Entwicklung

Die Metapher:

- Die Benutzt-Beziehung zwischen Klassen wird als Vertragsverhältnis zwischen **Klient** (engl.: Client) und **Dienstleister** (engl.: Supplier) interpretiert.
- In diesem Vertrag wird formuliert,
  - welche **Vorbedingungen** ein Klient einhalten muss,
  - damit der Lieferant seine Dienstleistung erbringt
  - und dies über **Nachbedingungen** auch garantiert.
- Der Begriff **Vertragsmodell** ist unsere deutsche Übersetzung des englischen **Design by Contract**, wie es von Bertrand Meyer geprägt wurde.

- Das Vertragsmodell wurde erstmalig in der Sprache **Eiffel** direkt umgesetzt.

Ein erstes Beispiel in Eiffel

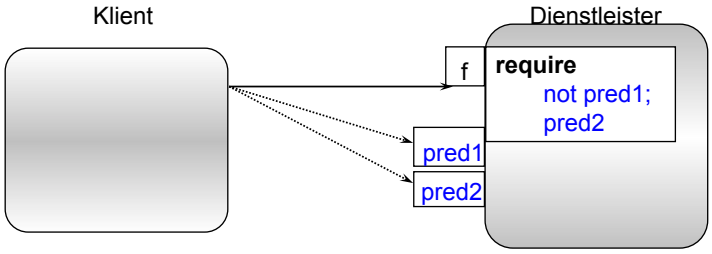


- Assume for example [...] the task of inserting a certain element into a dictionary [...] of bounded capacity. http://www.eiffel.com/values/design-by-contract/introduction/

```
put (x: ELEMENT; key: STRING) is
                -- Insert x so that it will be retrievable through key.
        require
                count <= capacity
                not key.empty
        do
                ... Some insertion algorithm ...
        ensure
                has (x)
                item (key) = x
                count = old count + 1
        end
```

- Man beachte Eiffels Schlüsselwort `old`, mit welchem man sich auf das Ergebnis von `count` vor dem Aufruf der Operation `put` beziehen kann!
- Die erste Vorbedingung sollte besser `count < capacity` lauten ☺

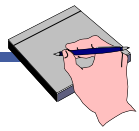
Der Klient muss seine Vorbedingungen prüfen können



Vertragsmodell und Zusicherungen:

- Damit ein Vertrag sinnvoll ist, muss der Klient die Einhaltung der Vorbedingungen einer Operation überprüfen können.
- Daher müssen alle in einer Vorbedingung verwendeten Terme geeignet als Prädikate (sondierende Operationen) an der Schnittstelle des Dienstleisters sichtbar sein.
- Insbesondere dürfen keine (privaten) Exemplarvariablen im Vertrag auftreten, da ein Klient keinen Zugriff darauf hat!

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* (aber nicht während der) Ausführung jeder Operation der Klasse gelten muss.
- **In der Praxis** findet man häufig Klassen-Invarianten, die sich auf (für Klienten irrelevante) **Implementationsdetails** beziehen, um die Klasse robust gegen zukünftige Änderungen/Erweiterungen zu machen.
- Sinnvolle Invarianten für die Klasse `ArrayTitelListe` aus SE1:
  - `_anzahlTitel <= _titelArray.length`
  - `keineNullBis(_titelArray, _anzahlTitel)`
  - `alleNullAb(_titelArray, _anzahlTitel)`

Vertragsmodell: Beispiel `Stack` (in Eiffel)

```
class Stack
 feature

   pop is
      require
         not empty
      do
         number_of_elements := number_of_elements - 1;
      ensure
         not full
      end;
      •••
 invariant
   empty implies (number_of_elements = 0)
end -- class Stack
```

Vorbedingung
(d.h. Voraussetzungen)

Nachbedingung
(d.h. Verpflichtungen)

Invariante
(d.h. Randbedingung)

Verträge unterstützen Software-Zuverlässigkeit

Klasseninvariante des **Lieferanten**:

- *Randbedingung* für Verträge zwischen Klient- und Lieferant-Objekt.

Vor- und **Nachbedingungen** einer **Routine**:

- *Vertrag* (für die Benutzung der Routine), der Klient- und Lieferant-Objekt bindet (bzw. ProgrammiererInnen).
- Ein Vertrag gilt pro Aufruf einer Operation.

Verpflichtungen **Rechte**

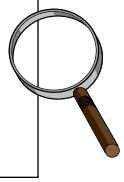
Wenn
der **Klient** die **Vorbedingungen** erfüllt,
dann
garantiert der **Lieferant** die **Nachbedingungen**.

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 verletzt sein.
- Private Methoden können aus Operationen aufgerufen werden, während die Invariante verletzt ist; daher herrscht zum Zeitpunkt des Betretens/Verlassens privater Methoden i.A. kein stabiler Zustand.

- 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.



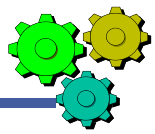
nach © Meyer

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.

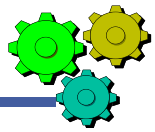
Manuelle Umsetzung des Vertragsmodells in Java



- 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 können ggf. unmittelbar vor dem Rücksprung geprüft werden; in SE2 verzichten wir jedoch darauf (siehe Folie 15).
    - Die Invarianten werden meist nur im Klassenkommentar notiert. Im Übungsbetrieb SE2 spielen Invarianten eine untergeordnete Rolle.

- 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



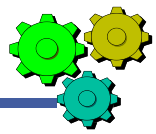
- Für die manuelle Überprüfung gibt es verschiedene Möglichkeiten:
  - Eine **spezifische Fehlerbehandlung für jeden Einzelfall** mit jeweils passendem Exception-Typ.
  - Eine zentrale Implementation, z.B. über Klassenmethoden einer `Contract`-Klasse wie im **Framework JWAM**:

```
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`



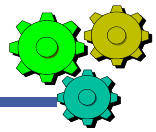
- Die **assert-Anweisung** von Java gibt es in zwei Varianten:

```
assert Condition ;
assert Condition : Description ;
```

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

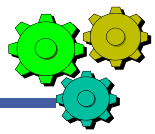
```
/**
 * Zahlt einen Betrag auf dieses Konto ein.
 * @require betrag >= 0
 */
public void zahleEin(int betrag)
{
    assert betrag >= 0 : "Vorbedingung verletzt: betrag >= 0";
    ...
}
```


Assertions in der VM aktivieren



- Normalerweise sind Assertions deaktiviert, d.h. `assert`-Anweisungen werden zur Laufzeit von der VM komplett ignoriert.
- Um Assertions zu aktivieren, muss man die VM speziell starten:
  `c:\Mein\Java\Projekt> java -ea MeineStartKlasse`
- Der Kommandozeilenparameter `-ea` steht dabei für „enable assertions“.
- Unter Eclipse kann man `-ea` unter folgendem Menüpunkt eintragen:
  Window/Preferences/Java/Installed JREs/Edit…/Default VM arguments

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 Vertragsmodell 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…

Warum überprüfen wir in SE2 keine Nachbedingungen?

- Nachbedingungen, die sich auf den alten Zustand eines Objekts vor dem Operationsaufruf beziehen, sind **in Java nicht formulierbar**.
  - Beispiel: „Der neue Kontostand ist gleich der alte Kontostand plus der eingezahlte Betrag.“ (Es gibt in Java kein Schlüsselwort `old`.)
- Bei trivialen Nachbedingungen lohnt sich eine Überprüfung oft nicht.
  - Beispiel: `return new X();` liefert niemals `null`.
- Nicht-triviale Nachbedingungen benötigen oft viel (duplizierten) Code.
  - **Lesbarkeit und Wartbarkeit** leiden.
  - Brutstätte für neue Fehler

Warum überprüfen wir in SE2 keine Nachbedingungen?

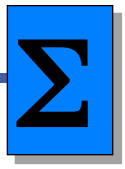
- Gut versteckte Fehler basieren oft auf **Missverständnissen** eines Klienten bezüglich des vom Dienstleister **angebotenen Vertrags**.
  - Die Überprüfung der Vorbedingungen kann *Programmierfehler* aufdecken, die an völlig anderen Stellen im Programm verortet sind.
  - Wenn man Nachbedingungen konsequent überprüft, findet man damit nur Fehler innerhalb derselben Methode; diese können durch *Code Reviews* und *Unit Tests* bereits sehr effektiv aufgespürt werden.
- Das **Kosten/Nutzen-Verhältnis** beim Überprüfen von Zusicherungen ist für Nachbedingungen deutlich ungünstiger als für Vorbedingungen.

- Die Überprüfung von Nachbedingungen ist nicht grundsätzlich schlecht/sinnlos! Wir haben uns für den Übungsbetrieb SE2 bewusst für eine einfache Richtlinie entschieden, damit wir uns im Laborbetrieb nicht in ausufernden Diskussionen verlieren, ob es im Einzelfall sinnvoll ist, eine Nachbedingung zu überprüfen oder nicht.

Nachbedingungen als Inspirationsquelle für Testfälle

- Nachbedingungen halten die Semantik einer Operation formal fest.
  - „Der neue Kontostand ist gleich der alte Kontostand plus der eingezahlte Betrag.“
  - „Nachdem ein Element in die Liste eingefügt wurde, ist es in der Liste enthalten.“
  - „Nachdem ein Element aus der Menge entfernt wurde, ist es nicht mehr in der Menge enthalten.“
- Wir kennen bereits ein adäquates Mittel, um die Semantik von Operationen systematisch zu überprüfen: Unit Tests.
- Für den Übungsbetrieb SE2 einigen wir uns auf folgendes Vorgehen:
  - **Vorbedingungen** werden **vom Dienstleister** am Anfang des Methodenrumpfs mit assert-Anweisungen **überprüft**.
  - **Nachbedingungen** werden *nicht* vom Dienstleister überprüft, sondern dienen **als Inspirationsquelle für Testfälle**.

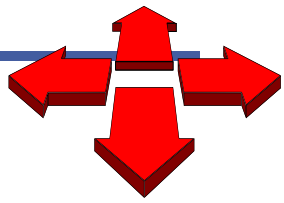
Zwischenergebnis Vertragsmodell



- Das **Vertragsmodell formalisiert** die **Benutzt-Beziehung** zwischen **Klient** und **Dienstleister**.
- 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) werden vom Dienstleister festgelegt.
- Die Zusicherungen sind Prädikate, die vom Klienten überprüft werden können. Sie enthalten **keine Implementationsdetails**.
- Wir werden das Vertragsmodell noch einmal im Zusammenhang mit **Vererbung** diskutieren.

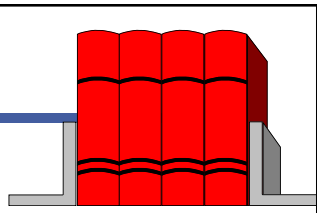
SE2 – OOPM – Teil 1 21

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.

Aus SE1: Wann ist Software überhaupt „korrekt“?

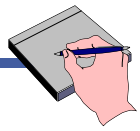
- Die **Korrektheit** von Software kann immer nur in Relation zu ihrer **Spezifikation** gesehen werden - eine Software-Einheit ist korrekt, wenn sie ihre Spezifikation erfüllt.
- Der formale Nachweis, dass eine Software-Einheit ihre Spezifikation erfüllt, ist sehr aufwendig und schwierig.
- Voraussetzung 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 somit der Begriff der **Validität**. (Gültigkeit)

Aus SE1: Statische und dynamische Tests



- Linz und Spillner unterscheiden in **Basiswissen Softwaretest** grundlegend zwischen statischen und dynamischen Tests.
- Ein **statischer Test** (häufig und u.E. treffender **statische Analyse** genannt) bezieht sich primär auf 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 Analyse

- Statische Tests/Analysen werden **nicht am laufenden System**, sondern **an seinen Dokumenten** (Quellcode und Dokumentation) ausgeführt.
- Sie sollen Fehler identifizieren, Architekturvorgaben überprüfen und die Qualität verbessern.
- Statische Tests/Analysen 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
    - Konformitätsprüfungen bezogen auf die Architektur

nach [Spillner, Linz]

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

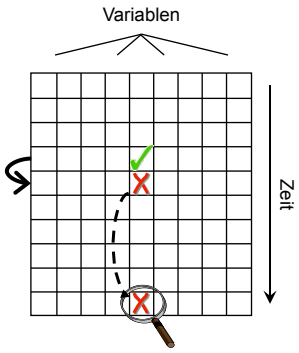
nach [Spillner, Linz]

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 erwartungskonform verhält.
  - **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

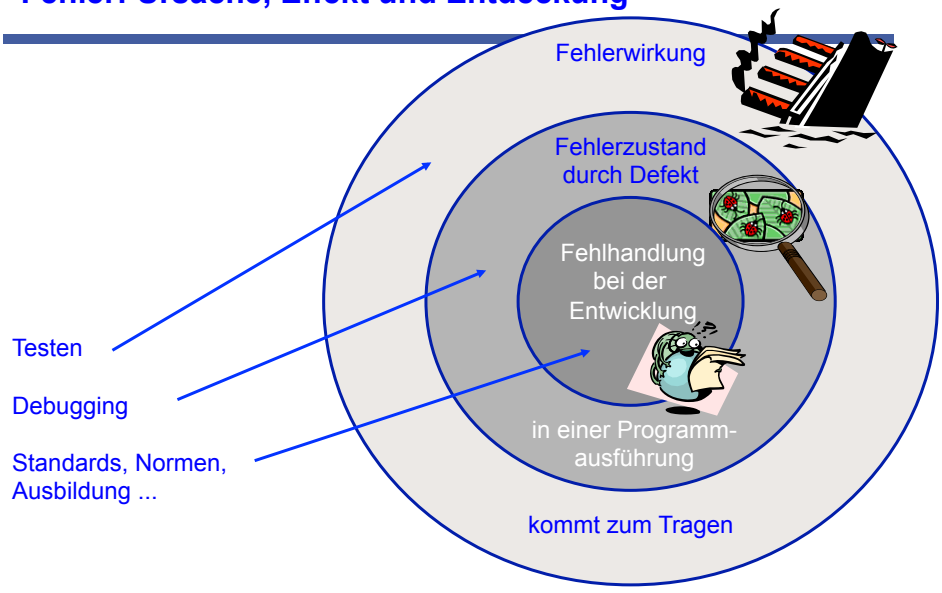
- Zeller differenziert Laufzeitfehler **bei der Suche nach Fehlern** in Software etwas genauer:
  - **defect** – der eigentliche **Defekt**, etwa eine fehlerhafte Anweisung in einer Quelltextzeile;
  - **infection** – die **Verfälschung** des Speicherzustandes aufgrund eines Defektes, der zur Ausführung 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.

SE2 – OOPM – Teil 1 31

Fehler: Ursache, Effekt und Entdeckung



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

SE2 – OOPM – Teil 1 32

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)

Grundsätze zum Testen (2)

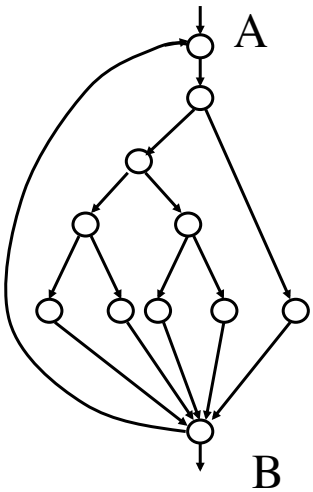
- **Mit dem Testen frühzeitig beginnen.**
  Testen ist keine späte *Phase* in der Softwareentwicklung, es soll damit so früh wie möglich begonnen werden. Durch frühzeitiges Prüfen (z.B. Reviews) parallel zu den konstruktiven Tätigkeiten werden Fehler(zustände) früher erkannt und somit Kosten gesenkt.
- **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.
- **Erfolgreiche Tests garantieren nicht Benutzbarkeit.**
  Ein System ohne Fehlerwirkungen bedeutet nicht, dass das System auch den Vorstellungen der späteren Nutzer entspricht.

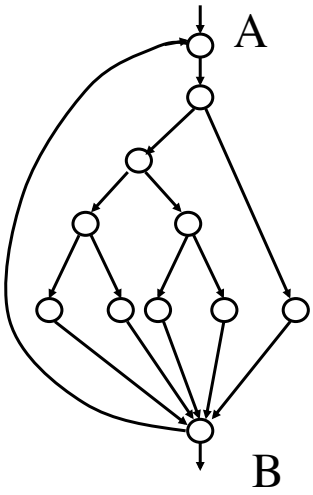
Austesten?

- 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:
  $5^1 + 5^2 + ... + 5^{18} + 5^{19} + 5^{20}$
- Wie lange dauert das Austesten bei 100.000 Tests pro Sekunde?



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:
  $5^1 + 5^2 + ... + 5^{18} + 5^{19} + 5^{20}$
- Wie lange dauert das Austesten bei 100.000 Tests pro Sekunde?
- Es sind 119.209.289.550.780 Testfälle
  Dauer: ca. 38 Jahre



Weshalb trotzdem testen?

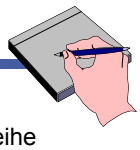
- Der **Quelltext kann leichter weiterentwickelt werden**, weil nach Veränderungen mit den Tests die Funktionstüchtigkeit getestet werden kann.
- Die **Debugging-Zeiten reduzieren sich**, weil durch die Tests Fehler schneller lokalisiert werden können.
- **Schnittstellen werden einfach**, da jeder Programmierer 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“** 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.

nach [Spillner, Linz]

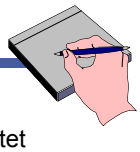
Aus SE1: Positives und negatives Testen



- 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).

nach [Spillner, Linz]

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

nach [Spillner, Linz]

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.

Testgetriebene Entwicklung

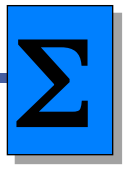
- **Agile Methoden** empfehlen, für jede Klasse eine eigene Testklasse zu schreiben.
- Beim sog. **Test First-Ansatz** sollen Testklassen **vor** den zu testenden Klassen geschrieben werden. Vor der Implementation der Methoden einer Klasse wird durch entsprechende Tests spezifiziert, welches Problem 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.

nach J. Link

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** (Programmierfehler, 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.

