Funktionen spezifizieren und testen

Software-Praktikum
Andreas Zeller, Universität des Saarlandes
mit Christian Lindig

Again, a test. We test whether we can evacuate 500 people from an Airbus A380 in 90 seconds. This is a test.

And: We test whether a concrete wall (say, for a nuclear reactor) withstands a plane crash at 900 km/h. Indeed, it does.

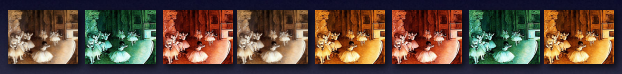
Edgar Degas: The Rehearsal. With a rehearsal, we want to check whether everything will work as expected. This is a test.

We can also test software this way. But software is not a planned linear show – it has a multitude of possibilities. So: if it works once, will it work again? This is the central issue of Testen – and of any verification method.

Software ist vielfältig

We can also test software this way. But software is not a planned linear show – it has a multitude of possibilities. So: if it works once, will it work again? This is the central issue of Testen – and of any verification method.

Software ist vielfältig



The problem is: There are many possible executions. And as the number grows…

Software ist vielfältig

and grows…

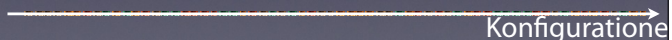
Software ist vielfältig

and grows…

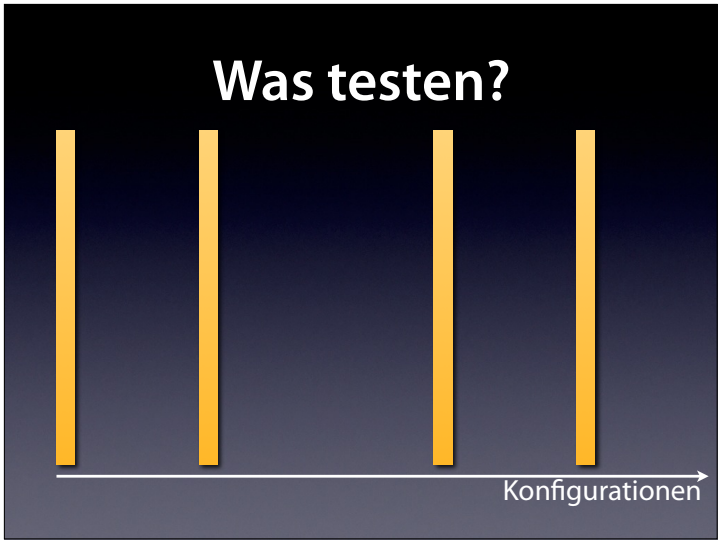
Software ist vielfältig

and grows…

Testen

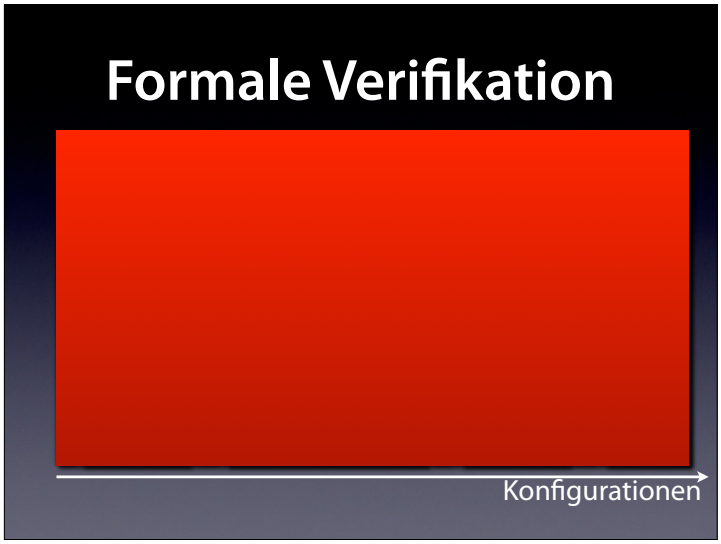


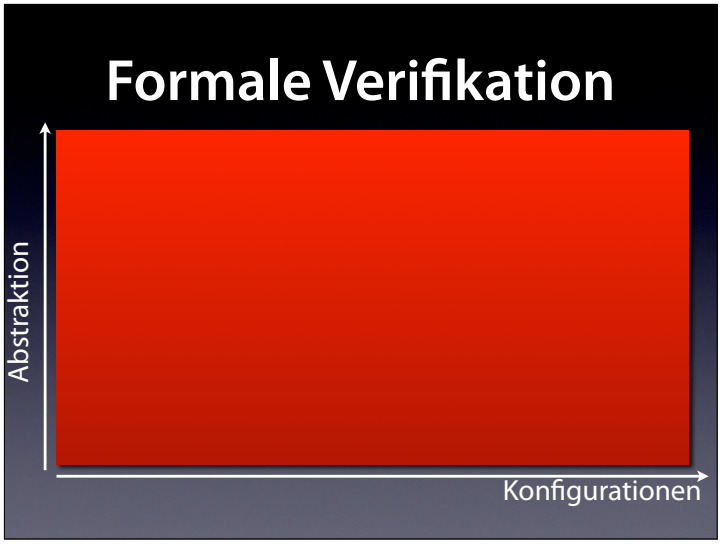
…you get an infinite number of possible executions, but you can only conduct a finite number of tests.

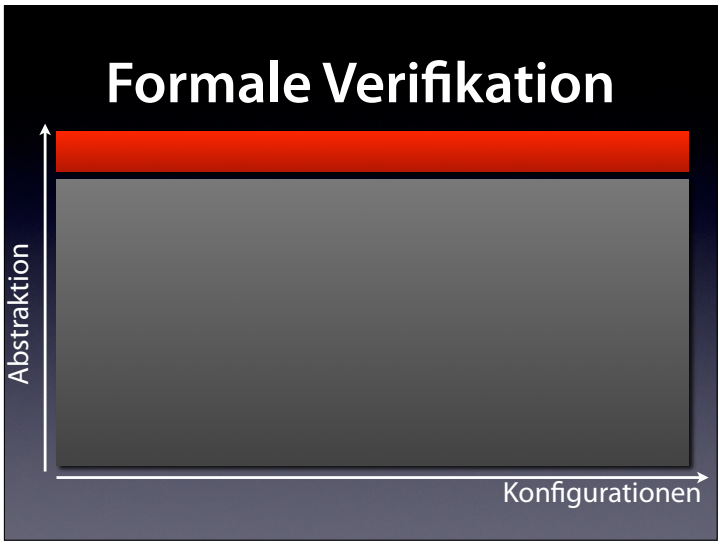


So, how can we cover as much behavior as possible?

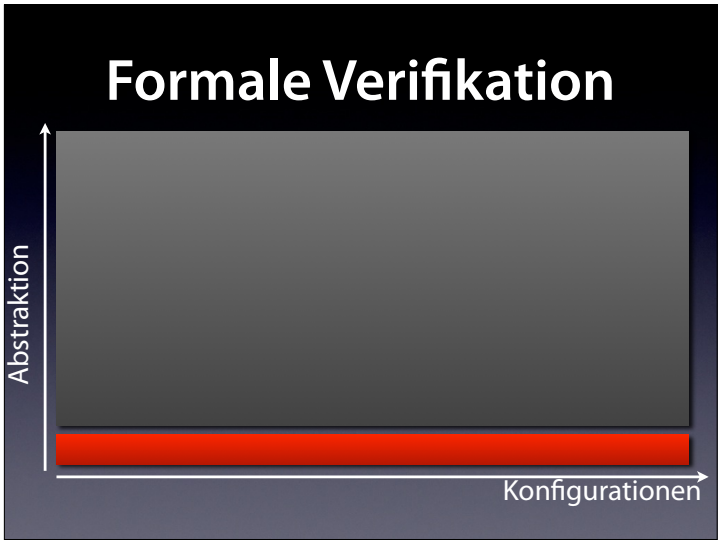
But still, Testen suffers from what I call Dijkstra's curse – a double meaning, as it applies both to Testen as to his famous quote. Is there something that can find the **absence** of errors?



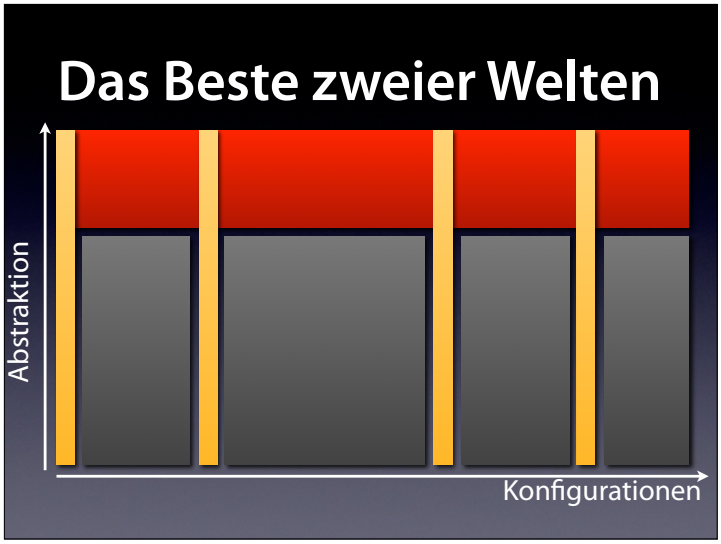




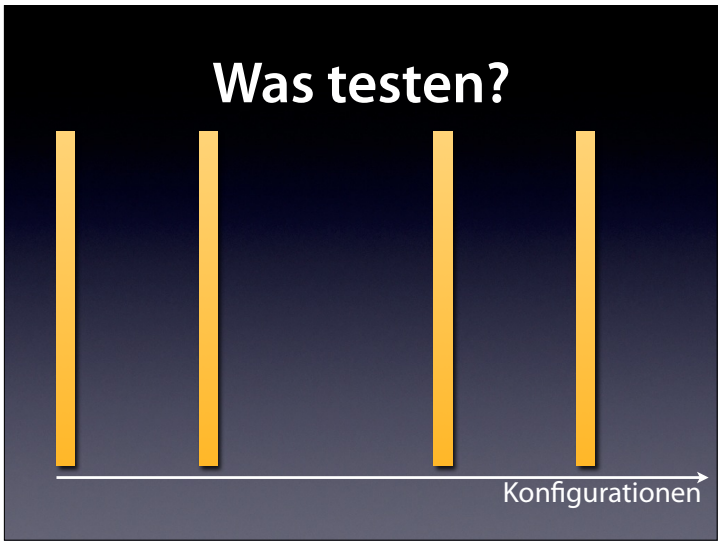
Areas missing might be: the operating system, the hardware, all of the world the system is embedded in (including humans!)



Areas missing might be: the operating system, the hardware, all of the world the system is embedded in (including humans!)



We might not be able to cover **all** Abstraktion levels in **all** Konfigurationens, but we can do our best to cover as much as possible.



So, how can we cover as much behavior as possible?

So, how can we cover as much behavior as possible?

Spezifikation

Spezifikationsverfahren werden während der Software-Entwicklung eingesetzt, um die *Semantik ausgewählter Funktionen* zu beschreiben.

In der Regel geschieht dies

- ohne Beteiligung des Kunden
- wenn der Entwurf (Systemmodell, Sequenzdiagramme) feststehen
- für ausgewählte (kritische, zentrale) Funktionen

Anforderungen

Jede Spezifikation soll

- *vollständig* sein – jeder Aspekt des Systemverhaltens wird abgedeckt
- *widerspruchsfrei* sein – damit klar ist, was implementiert werden soll
- auch *unvorhergesehene Umstände* beschreiben, um die Robustheit zu steigern.

Vorsicht: auch eine in sich perfekte Spezifikation kann im Widerspruch zu den Absichten des Auftraggebers stehen.

Verfahren

Man unterscheidet folgende *Spezifikationsverfahren*:

Informale Spezifikation in Prosa (natürlicher Sprache)

Formale Spezifikation in spezieller Modellierungssprache (VDM, Z)

Exemplarische Spezifikation mit Testfällen

Informale Spezifikation

Für jede Funktion wird in kurzer Prosa beschrieben, was sie tut. Die Beschreibung sollte zumindest enthalten:

- den Zweck der Funktion
- die Rolle der Parameter und des Rückgabewertes
- ggf. Seiteneffekte
- Verhalten bei Fehlern

✔ Weitverbreitetes Spezifikationsverfahren

✔ Gut für *Dokumentation geeignet*

✘ Unexakt, oft unvollständig

✘ Einhaltung der Spezifikation schwer nachweisbar

Spezifikation einer Prozess-Steuerung

Einfache informale Spezifikation, gegliedert nach Klassen und Methoden:

Klasse **`Control`**:

- `int Control.get_temperature()` liefert die Temperatur des Reaktors in Grad Celsius zurück.
- `boolean Control.(input|output|emergency)_valve_open()` liefert `true`, wenn das betreffende Ventil geöffnet ist; sonst `false`.
- `void Control.set_(input|output|emergency)_valve (boolean valve_open)` öffnet das betreffende Ventil, wenn `valve_open` den Wert `true` hat; ansonsten wird das Ventil geschlossen.

Hier werden mehrere ähnliche Methoden zu einem Muster zusammengefasst. Dies vermeidet Cut/Copy/Paste von immer wiederkehrenden Abschnitten.

UNIX-Funktion open

Das UNIX-Referenzhandbuch beschreibt für jede bereitgestellte Funktion, was sie tut:

Name open—open and possibly create a file or device

Synopsys

```
#include <sys/types.h> […]

int open(const char *pathname, int flags);
```

Description The open() system call is used to convert a *pathname* into a file descriptor (a small, non-negative integer for use in subsequent I/O as with read, write, etc.). […] *flags* is one of O_RDONLY, O_WRONLY or O_RDWR which request opening the file read-only, write-only or read/write, respectively. […]

Return Value open returns the new file descriptor, or -1 if an error occurred (in which case, `errno` is set appropriately). […]

Bewährtes und verbreitetes Schema!

Perl-Inkrement-Operator

Abschreckendes Beispiel: Spezifikation des Perl-++-Operators

> *The autoincrement operator [++] has a little extra built-in magic. If you increment a variable that is numeric, or that has ever been used in a numeric context, you get a normal increment. If, however, the variable has only been used in string contexts since it was set, has a value that is not the null string, and matches the pattern* `/^[a-zA-Z]*[0-9]*$/`*, the increment is done as a string, preserving each character within its range, with carry:*
>
> ```
> print ++($foo = '99');  # prints '100'
> print ++($foo = 'a9');  # prints 'b0'
> print ++($foo = 'Az');  # prints 'Ba'
> print ++($foo = 'zz');  # prints 'aaa'
> ```

- Was ist ein Kontext?
- Werden Unicode-Strings ebenfalls inkrementiert? Wie?
- Was ist der Wert von `$foo`?

Wenn Sie jemals ++ implementieren müssen: Viel Spass!

Verifikation und Validierung

Verifikation und Validierung prüfen beide, ob ein Produkt seine Spezifikation erfüllt.

Verifikation garantiert die Erfüllung der Spezifikation.

Validierung prüft die Erfüllung der Spezifikation stichprobenhaft.

Beispiel: *Produktion von Seilen* –
Verifikation prüft Zugfestigkeit jedes Seiles,
Validierung nur die ausgewählter Seile.

Verifikation von Software verlangt formale Beweise;
Validierung kann durch *Test* der Software erreicht werden.

Formale Spezifikation

Mittels einer *formalen Beschreibungssprache* wird die Semantik der Funktionen exakt festgelegt.

✔ Exakte Beschreibung der Semantik

✔ Ausführbare Spezifikationssprache kann als Prototyp dienen

✔ Möglichkeit des Programmbeweises

✘ Erhöhte Anforderungen an Verwender

✘ Aufwändig

Spezifikation mit Bedingungen

In der Praxis werden komplexe Funktionen über *Bedingungen* spezifiziert:

Vorbedingungen für Voraussetzungen

Nachbedingungen für Wirkung

Invarianten für Unveränderliches

Beispiel Modulo-Division $z = x \bmod y$, die für negative Argumente oft nicht eindeutig definiert ist:

Vorbedingung $y \neq 0$

Nachbedingung $z = x - y\lfloor x/y \rfloor$

Vorbedingungen

beschreiben die *Voraussetzungen* zur Ausführung einer Funktion. Hierzu gehören:

- Aussagen über die Eingabeparameter
- Aussagen über den Programmzustand (sichtbar und unsichtbar)

Beispiel System-Call
`int read(int fd, void *buf, size_t count)`:

- `fd` ist ein von `open()` oder `dup()` erhaltenes File-Handle.
- $0 \leq$ `count` $<$ `SSIZE_MAX`

"`read()` attempts to read up to `count` bytes from file descriptor `fd` into the buffer starting at `buf`."

Nachbedingungen

beschreiben die *Wirkung*, den die Ausführung einer Funktion bewirkt. Hierzu gehören:

- Aussagen über die Ausgabeparameter
- Aussagen über den Programmzustand

jeweils in Abhängigkeit vom Vorzustand und den Eingabeparametern, hier für `read()`:

- Bei Erfolg, wird die Anzahl $n \geq 0$ gelesener Bytes zurückgegeben.
- Der mit `fd` assoziierte interne Cursor ist um n Bytes vorgerückt.
- Im Fehlerfall ist das Ergebnis -1 und `errno` gesetzt.

Invarianten

sind Aussagen, die *vor* und *nach* jeder Funktion gelten – typischerweise über Objekt- und Klassenzustände

- `read()` verändert den Zustand des Filesystems nicht.

Relativ uninteressant für `read()`

Für eine sortierte Datenstruktur wäre der Erhalt der Sortierung nach einer Einfüge- oder Lösch-Operation eine wichtige Invariante.

Beispiel: Ein Editor in Z

Ein Texteditor speichert den Text in zwei Listen:
left steht *vor* der aktuellen Cursor-Position, *right* ist *danach*.

Die Gesamtlänge des Textes darf *maxsize* nie überschreiten.

Zustand und Invariante werden durch ein *Schema* der Modellierungssprache Z beschrieben:

Editor

$$left, right : TEXT$$

$$\#(left \frown right) \leq maxsize$$

$\frown$: Konkatenation zweier Folgen
$\#$: Anzahl der Elemente

Beispiel: Ein Editor in Z (2)

Wir modellieren das Einfügen eines einzelnen Zeichens:

Insert

$$\Delta Editor$$
$$ch? : CHAR$$

$$ch? \in printing$$
$$left' = left \frown \langle ch? \rangle$$
$$right' = right$$

$\Delta Editor$: Operations-Schema auf *Editor*
$ch?$: Eingabevariable
$ch? \in printing$: Vorbedingung („Zeichen muss druckbar sein“)
$left'$, $right'$: Zustand *nach* der Operation

Herausforderungen

Beweis zentraler Eigenschaften. Für jede Operation muss bewiesen werden, dass sie die Bedingungen einhält – etwa die Invariante

$$\#(left \frown right) \leq maxsize$$

Korrekte Konkretisierung. Beim Umsetzen in die endgültige Programmiersprache muss sichergestellt sein, dass Semantik (und somit die bewiesenen Eigenschaften) erhalten bleiben.

Ergebnis: *korrekte Software!*

Statische vs. dynamische Prüfung

An Stelle der vollständigen Verifikation können auch *Laufzeitprüfungen zur Validierung* treten.

Beispiel: Prüft eine Funktion bei jedem Aufruf, ob ihre Vor- und Nachbedingungen erfüllt sind, ist per Definition die Korrektheit der Funktion gegeben – *wenn* sie ein Ergebnis liefert.

Zusicherung

In der Praxis werden Bedingungen häufig zur Laufzeit geprüft!

Praktische Anwendung mit *Zusicherungen* (assertions) – `assert(x)` bricht die Ausführung ab, wenn x unwahr ist

```
static void delay(const char *fmt, ...)
{
    va_list ap;
    char buf[1024];
    assert(fmt != NULL);
    buf[sizeof(buf)-1] = 0;
    va_start(ap, fmt);
    vfprint(NULL, buf, fmt, ap);
    va_end(ap);
    delayed = append(string(buf),delayed);
    assert(buf[sizeof(buf)-1]==0);
}
```

Zusicherungen können komplett abgeschaltet werden, was in der Regel eine schlechte Idee ist.

Gute Zusicherungen

- prüfen Vor- und Nachbedingungen sowie Invarianten
- nutzen spezielle Methoden (repOK()), um Invarianten sicherzustellen

Eine Time-Klasse

```
class Time {
public:
    int hour();     // 0..23
    int minutes();  // 0..59
    int seconds();  // 0..60 (incl. leap seconds)

    void set_hour(int h);
    …
}
```

Gültige Zeiten: 00:00:00 bis 23:59:60

Konsistenz sichern

```
void Time::set_hour(int h)
{
    // Vorbedingung
    assert (0 <= hour() && hour() <= 23) &&
           (0 <= minutes() && minutes() <= 59) &&
           (0 <= seconds() && seconds() <= 60);
    …
    // Nachbedingung
    assert (0 <= hour() && hour() <= 23) &&
           (0 <= minutes() && minutes() <= 59) &&
           (0 <= seconds() && seconds() <= 60);
}
```

Konsistenz sichern

```
bool Time::repOK()
{
    return (0 <= hour() && hour() <= 23) &&
           (0 <= minutes() && minutes() <= 59) &&
           (0 <= seconds() && seconds() <= 60);
}

void Time::set_hour(int h)
{
    assert (repOK());  // Vorbedingung
    …
    assert (repOK());  // Nachbedingung
}
```

Konsistenz sichern

```
bool Time::repOK()
{
    return (0 <= hour() && hour() <= 23) &&
           (0 <= minutes() && minutes() <= 59) &&
           (0 <= seconds() && seconds() <= 60);
}
```

repOK() ist die *Invariante* eines Time-Objekts:

- gilt *vor* jeder öffentlichen Methode
- gilt *nach* jeder öffentlichen Methode

Konsistenz sichern

```
bool Time::repOK()
{
    return (0 <= hour() && hour() <= 23) &&
           (0 <= minutes() && minutes() <= 59) &&
           (0 <= seconds() && seconds() <= 60);
}

void Time::set_hour(int h)
{
    assert (repOK());
    …
    assert (repOK());
}
```

genauso für set_minute(), set_seconds(), usw.

Fehler lokalisieren

- Vorbedingung schlägt fehl = Fehler vor Aufruf
- Nachbedingung schlägt fehl = Fehler in Aufruf
- Alle Zusicherungen ok = kein Fehler

```
void Time::set_hour(int h)
{
    assert (repOK());
    …
    assert (repOK());
}
```

Komplexe Invarianten

```
class RedBlackTree {
  …
  boolean repOK() {
      assert (rootHasNoParent());
      assert (rootIsBlack());
      assert (redNodesHaveOnlyBlackChildren());
      assert (equalNumberOfBlackNodesOnSubtrees());
      assert (treeIsAcyclic());
      assert (parentsAreConsistent());

      return true;
  }
}
```

Exemplarische Spezifikation

Durch *Testfälle* werden *Beispiele* für das Zusammenspiel der Funktionen samt erwarteter Ergebnisse beschrieben.

✔ Formales (da am Code orientiertes) Spezifikationsverfahren, dennoch leicht verständlich

✔ Nach der Implementierung dienen die Testfälle zur Validierung

✔ Messbare Qualitätssteigerung bei geringem Overhead

✘ Nur exemplarische Beschreibung (und daher nur Validierung) des Verhaltens; muss mit zumindest informaler Spezifikation ergänzt werden.

Exemplarische Spezifikation (2)

Im *Extreme Programming* gilt der Leitsatz, dass so *früh wie möglich* getestet werden soll:

- Die Testfälle werden bereits *vor der Implementierung* erstellt
- Tritt ein neuer, noch nicht abgedeckter Fehler auf, wird vor der Fehlersuche *ein Testfall erstellt,* der den Fehler reproduziert.

Die Testfälle werden so Teil der Spezifikation!

Um Umstrukturierung (Refactoring) zu erleichtern, werden die Tests *automatisiert;* die Programmierer erstellen, verwalten die Tests selbst und führen sie auch aus (etwa nach jeder Änderung).

Automatisches Testen mit JUnit

JUnit von Kent Beck und Erich Gamma ist ein Testrahmen für Regressionstests von Java-Komponenten.

Ziel von JUnit ist, die Produktion hochwertigen Codes zu beschleunigen.

Analog: *CPPUnit* für C++-Programme

Testfälle

JUnit stellt *Testfälle* (`Testcase`) bereit, organisiert nach dem Command-Pattern.

Ein Testfall besteht aus einer Menge von `testXXX()`-Methoden, die jeweils einen bestimmten Test realisieren; mit der ererbten `assertTrue()`-Methode werden erwartete Eigenschaften sichergestellt.

Zusätzlich gibt es `setUp()` zum Initialisieren einer (Test-)Umgebung *(Fixture)* sowie `tearDown()` zum Freigeben der Testumgebung.

Testsuiten

Die Tests eines Testfalls werden in einer *Testsuite* (`TestSuite`) zusammengefasst, die von der Methode `suite()` zurückgegeben werden. Testsuiten können ebenfalls Testsuiten enthalten (`Composite`-Pattern).

Testen eines Warenkorbs

Die Klasse `ShoppingCart` (Warenkorb; hier nicht angegeben) enthält Methoden zum Hinzufügen und Löschen von Produkten sowie zum Abfragen der Produktanzahl und des Gesamtpreises.

Wir implementieren einen Testfall als Klasse `ShoppingCartTest`, der die Funktionalität der Klasse testet.

Teil 1: Initialisierung

enthält *Konstruktor* sowie *Erzeugen* und *Zerstören* der Testumgebung

```
import junit.framework.Test;
import junit.framework.TestCase;
import junit.framework.TestSuite;

public class ShoppingCartTest extends TestCase {

    private ShoppingCart _bookCart;

    // Neuen Test erzeugen
    public ShoppingCartTest(String name) {
        super(name);
    }
```

Initialisierung (2)

```
// Testumgebung erzeugen
// Wird vor jeder testXXX()-Methode aufgerufen
protected void setUp() {

    _bookCart = new ShoppingCart();

    Product book =
        new Product("Extreme Programming", 23.95);
    _bookCart.addItem(book);
}

// Testumgebung wieder freigeben
protected void tearDown() {
    _bookCart = null;
}
```

Teil 2: Tests

Jeder Test wird als Methode `public void testXXX()` realisiert.

Ein Test führt einige Methoden aus und prüft dann, ob der Zustand den Erwartungen entspricht. Wenn nicht, gibt es einen Fehler.

Beispiel: Test auf leeren Warenkorb. Erst wird der Warenkorb geleert, dann wird geprüft, ob er auch tatsächlich leer ist.

```
// Test auf leeren Warenkorb
public void testEmpty() {
    _bookCart.empty();
    assertTrue(_bookCart.isEmpty());
}
```

Tests (2)

Wir benutzen die von TestCase ererbten Hilfsmethoden:

`fail(msg)` – meldet einen Fehler namens `msg`

`assertTrue(msg, b)` – meldet einen Fehler, wenn Bedingung b unwahr ist

`assertEquals(msg,` v_1**,** v_2**)** – meldet einen Fehler, wenn $v_1 \neq v_2$

`assertEquals(msg,` v_1**,** v_2**,** ϵ**)** – meldet einen Fehler, wenn $| v_1 - v_2 | > \epsilon$

`assertNull(msg, object)` – meldet einen Fehler, wenn `object` nicht null ist

`assertNonNull(msg, object)` – meldet einen Fehler, wenn `object` null ist

`msg` kann auch weggelassen werden.

Funktioniert das Hinzufügen?

```
// Test auf Hinzufügen
public void testProductAdd() {
    Product book = new Product("Refactoring", 53.95);
    _bookCart.addItem(book);

    double expectedBalance = 23.95 + book.getPrice();
    double currentBalance = _bookCart.getBalance();
    double tolerance = 0.0;
    assertEquals(expectedBalance, currentBalance,
                 tolerance);

    int expectedItemCount = 2;
    int currentItemCount = _bookCart.getItemCount();
    assertEquals(expectedItemCount, currentItemCount);
}
```

Funktioniert das Löschen?

```
// Test auf Löschen
public void testProductRemove()
    throws ProductNotFoundException {
    Product book =
        new Product("Extreme Programming", 23.95);
    _bookCart.removeItem(book);

    double expectedBalance = 23.95 - book.getPrice();
    double currentBalance = _bookCart.getBalance();
    double tolerance = 0.0;
    assertEquals(expectedBalance, currentBalance,
                 tolerance);

    int expectedItemCount = 0;
    int currentItemCount = _bookCart.getItemCount();
    assertEquals(expectedItemCount, currentItemCount);
}
```

Gibt es korrekte Fehlerbehandlung?

```
// Test auf Entfernen eines unbekannten Produkts
public void testProductNotFound() {
    try {
        Product book =
            new Product("Ender's Game", 4.95);
        _bookCart.removeItem(book);
        fail("Should raise a ProductNotFoundException");
    }
    catch(ProductNotFoundException pnfe) {
        // Test sollte stets hier entlang laufen
    }
}
```

Teil 3: Testsuite

Die Klasse wird mit einer Methode `suite()` abgeschlossen, die die einzelnen Testfälle zu einer Testsuite zusammenfasst. Dies geschieht gewöhnlich über Reflection – alle Methoden der Form `testXXX()` werden Teil der Testsuite.

```
// Testsuite erstellen
public static Test suite() {

    // Hier: Alle testXXX()-Methoden hinzufügen (über Reflection)
    TestSuite suite = new TestSuite(ShoppingCartTest.class);

    // Alternative: Methoden einzeln hinzufügen (fehleranfällig)
    // TestSuite suite = new TestSuite();
    // suite.addTest(new ShoppingCartTest("testEmpty");
    // suite.addTest(new ShoppingCartTest("testProductAdd");
    // suite.addTest(new ShoppingCartTest("testProductRemove");
    // suite.addTest(new ShoppingCartTest("testProductNotFound");

    return suite;
}
```

Teil 4: Hilfen

Schließlich müssen wir dem Testfall noch einen Namen geben (`toString()`). Die Hauptmethode `main()` ruft ein GUI für genau diesen Testfall auf.

```
  // String-Darstellung dieses Testfalls zurückgeben
  public String toString() {
      return getName();
  }
  // Hauptmethode: Ruft GUI auf
  public static void main(String args[]) {
      String[] testCaseName =
          { ShoppingCartTest.class.getName() };
      // junit.textui.TestRunner.main(testCaseName);
      junit.swingui.TestRunner.main(testCaseName);
  }
}
```

Damit ist die Klasse `ShoppingCartTest` vollständig.

Annotationen

- JUnit 4 und später:
  Tests werden mit "@Test" markiert:

  ```
  @Test
  public void testProductNotFound() { … }
  ```

- Analog: Setup mit "@Before"
- Analog: Teardown mit "@After"

Test ausführen

Ist die Implementierung abgeschlossen, kann der Testfall zur *Validierung* benutzt werden – indem er ausgeführt wird.

Das Ausführen eines Testfalls geschieht einfach über die `main()`-Methode, die (hier) eine graphische Oberfläche aufruft:

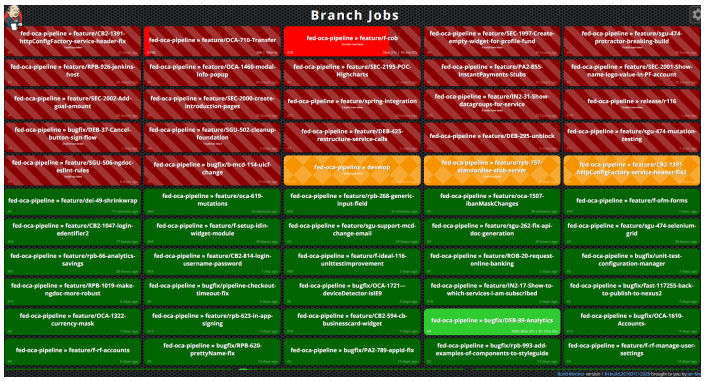
```
$ java ShoppingCartTest
```

Eine komplette TestSuite (aus mehreren Testfällen) wird ebenso ausgeführt:

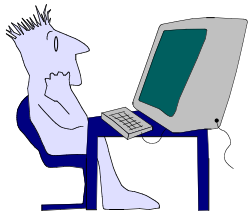
```
$ java EcommerceTestSuite
```

Test ausführen (2)

Die Testergebnisse werden im Fenster angezeigt:

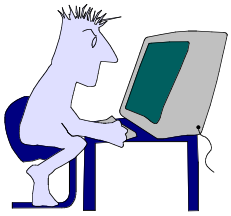


Wer testet?



Entwickler

- *versteht* das System
- wird *vorsichtig* testen
- will *Code* liefern



Unabhängiger Tester

- muss System *lernen*
- will *Fehler* aufdecken
- will *Qualität* liefern

From Pressman, “Software Engineering – a practitioner’s approach”, Chapter 13

Der beste Tester

A good tester should be creative and destructive – even sadistic in places.
– Gerald Weinberg, “The psychology of computer programming”

Der Entwickler

The conflict between developers and testers is usually overstated, though.

Weinberg’s Gesetz

Ein Entwickler ist nicht geeignet, den eigenen Code zu testen.

Theory: As humans want to be honest with themselves, developers are blindfolded with respect to their own mistakes.
Evidence: “seen again and again in every project” (Endres/Rombach)
From Gerald Weinberg, “The psychology of computer programming”

Wie testen?

Welche Testfälle brauche ich?

- Die Testfälle sollten jede Methode der zu testenden Klasse *wenigstens einmal* aufrufen
- Enthält die Beschreibung der Methode unterschiedliches Verhalten für verschiedene Fälle, sollte *jeder Fall einzeln* getestet werden.
- Je mehr Tests, je besser :-)
- Automatisch generierte Trivial-Methoden (z.B. getter/setter) haben geringe Priorität

Wann muss ich neu testen?

Am besten *automatisch* – nach jeder Änderung!