

Kapitel 9b

Test-Automatisierung

Stand: 29.01.2015

Warum automatisierte Tests?

Automatisierte Modultests mit JUnit

Test First Development‘ und ‚Continuous Testing‘

Automatisierte Testerstellung mit T3

Test-Arten

- Modul-Test (Unit Test)

- ◆ vor "check in" geänderter source ins Projekt-Repository
- ◆ testet interne Funktion einer Komponente
- ◆ oft von Entwickler selbst durchgeführt

- Integrations-Test

- ◆ für jeden "build"!
- ◆ testet Details des Zusammenspiels von Systemkomponenten
- ◆ oft von System-Integratoren selbst durchgeführt

- System-Test

- ◆ am Ende einer Iteration
- ◆ testet Interaktion zwischen Akteuren und System
- ◆ oft von Testern durchgeführt die wenig / keine Interna kennen

- Regressions-Test

- ◆ Wiederholung von Modul- / Integrations- / System-Tests nach Änderungen
- ◆ sicherstellen, daß die "offensichtlich korrekte" Änderung bisheriges Verhalten nicht invalidiert

Fokus im Folgenden:
regressive
Modul-Tests

Warum schreibt niemand Tests?

- Tätigkeiten eines Programmierers

◆ Verstehen was man tun soll	5%
◆ Überlegen wie man's tun kann	10%
◆ Implementieren	20%
◆ Testen	5%
◆ Debugging	60%

- „Fixing a bug is usually pretty quick but finding it is a nightmare.“

- Also warum testen wir nicht mehr, um weniger Debuggen zu müssen?

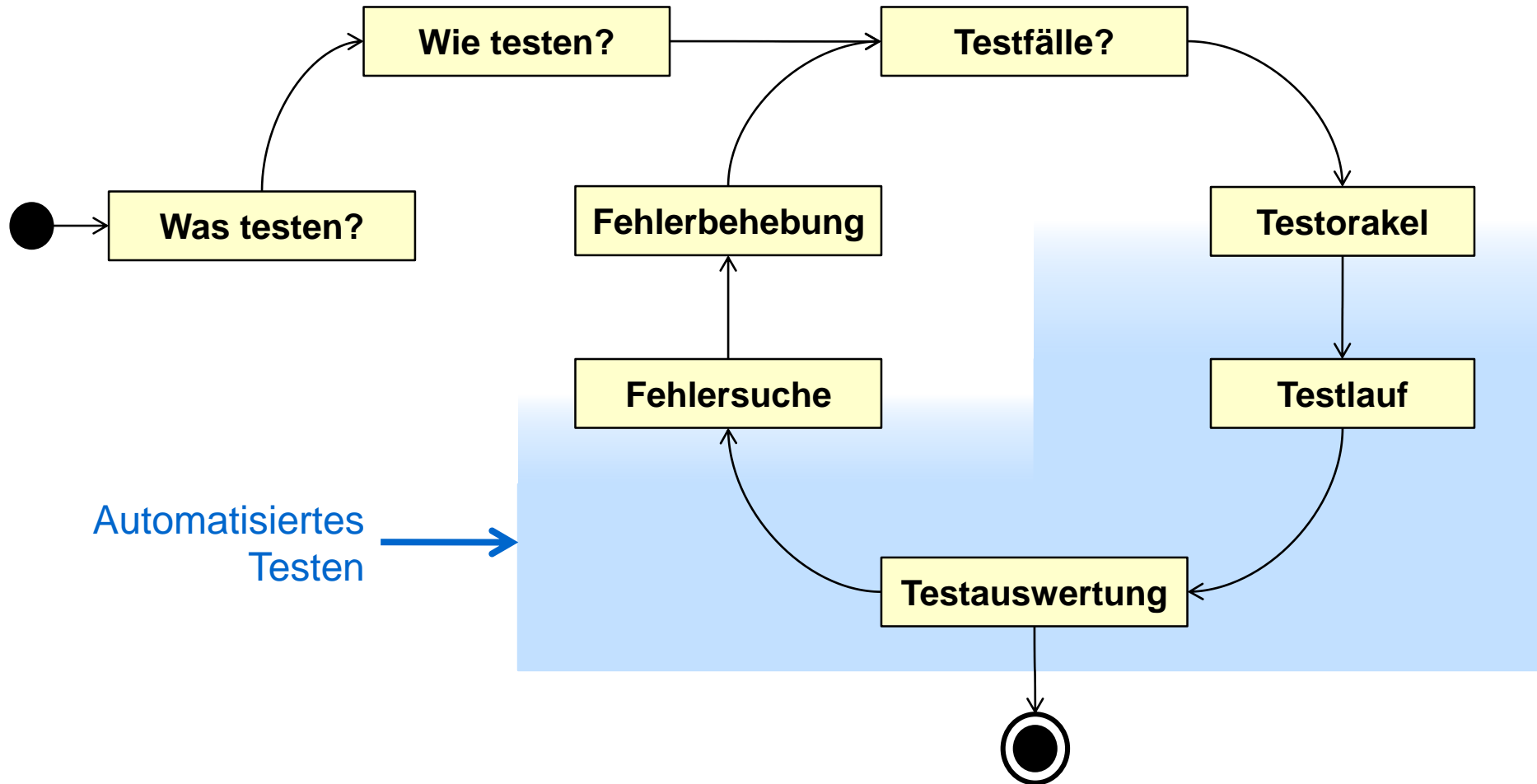
Warum schreibt niemand Tests?

- Tests mit „print“-Anweisungen im Programm
 - ◆ ständige Programmänderungen
 - ⇒ anderer Test = andere Print-statements
 - ⇒ Test deaktivieren = print-statements auskommentieren
 - ⇒ Test reaktivieren = Kommentare um print-statements löschen
 - ◆ langwierige Test-Auswertung
 - ⇒ ellenlange listings lesen
 - ⇒ überlegen, ob sie das widerspiegeln, was man wollte
 - ◆ Fazit
 - ⇒ es dauert alles viel zu lange
 - ⇒ Fehler werden eventuell doch übersehen
- Lehre
 - ◆ Tests müssen modular sein!
 - ⇒ ausserhalb der zu testenden Klasse
 - ◆ Tests müssen sich selbst auswerten!!!
 - ⇒ Testprogramm vergleicht tatsächliche Ergebnisse mit erwarteten Ergebnissen

Nutzen automatischer Tests

- Geringerer Aufwand
 - ◆ Tests zu schreiben
 - ◆ Tests zu warten
 - ◆ Tests zu aktivieren / deaktivieren
 - ◆ Tests zu komponieren
 - ◆ Tests durchzuführen und auszuwerten!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!
 - Testen in kürzeren Abständen möglich
 - ◆ Weniger Fehlerquellen zwischen Tests
 - ◆ Erinnerung was man verändert hat ist noch da
- Weniger Fehler
- Schnellere Identifikation der Fehlerursache
- Schnellere Programmentwicklung!!!

Automatisiertes Testen im Testzyklus



- Warum automatisierte Tests

- Das Junit-Framework

- Einführung

- ◆ Beispiel

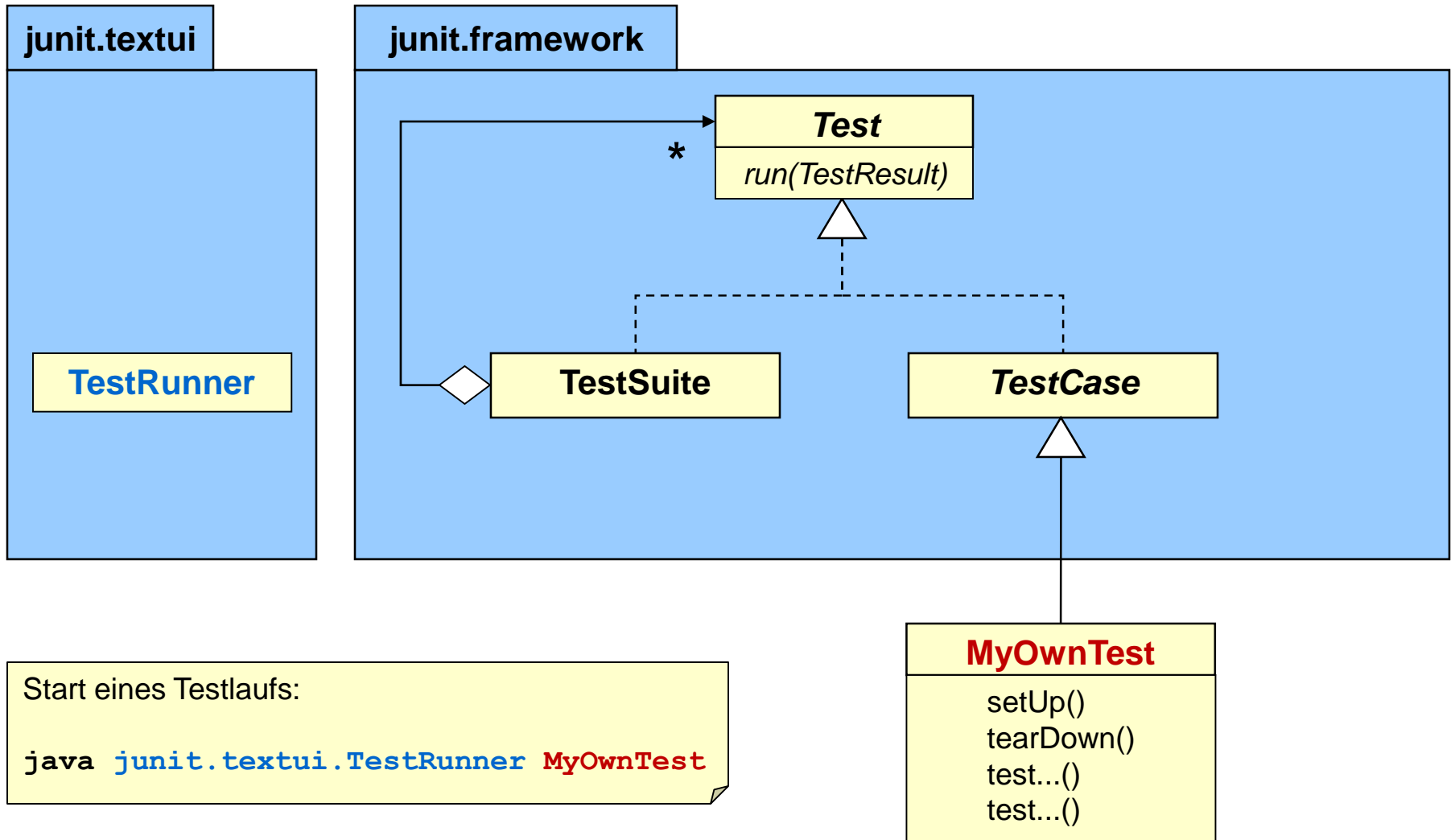
- ◆ Testen von GUIs

- Empfehlungen

Ziele von JUnit

- Das Framework muss
 - ◆ Testerstellungsaufwand auf das absolut Nötige reduzieren
 - ◆ leicht zu erlernen / benutzen sein
 - ◆ doppelte Arbeit vermeiden
 - Tests müssen
 - ◆ wiederholt anwendbar sein
 - ◆ separat erstellbar sein → getrennt vom zu testenden Code
 - ◆ inkrementell erstellbar sein → „Testsuites“
 - ◆ frei kombinierbar sein
 - ◆ auch von anderen als dem Autor durchführbar sein
 - ◆ auch von anderen als dem Autor auswertbar sein
 - Testdaten müssen
 - ◆ wiederverwendbar sein (Testdatenerstellung ist meist aufwendiger als der Test selbst)
- Erleichterung der Test-Erstellung, -Durchführung und –Auswertung!

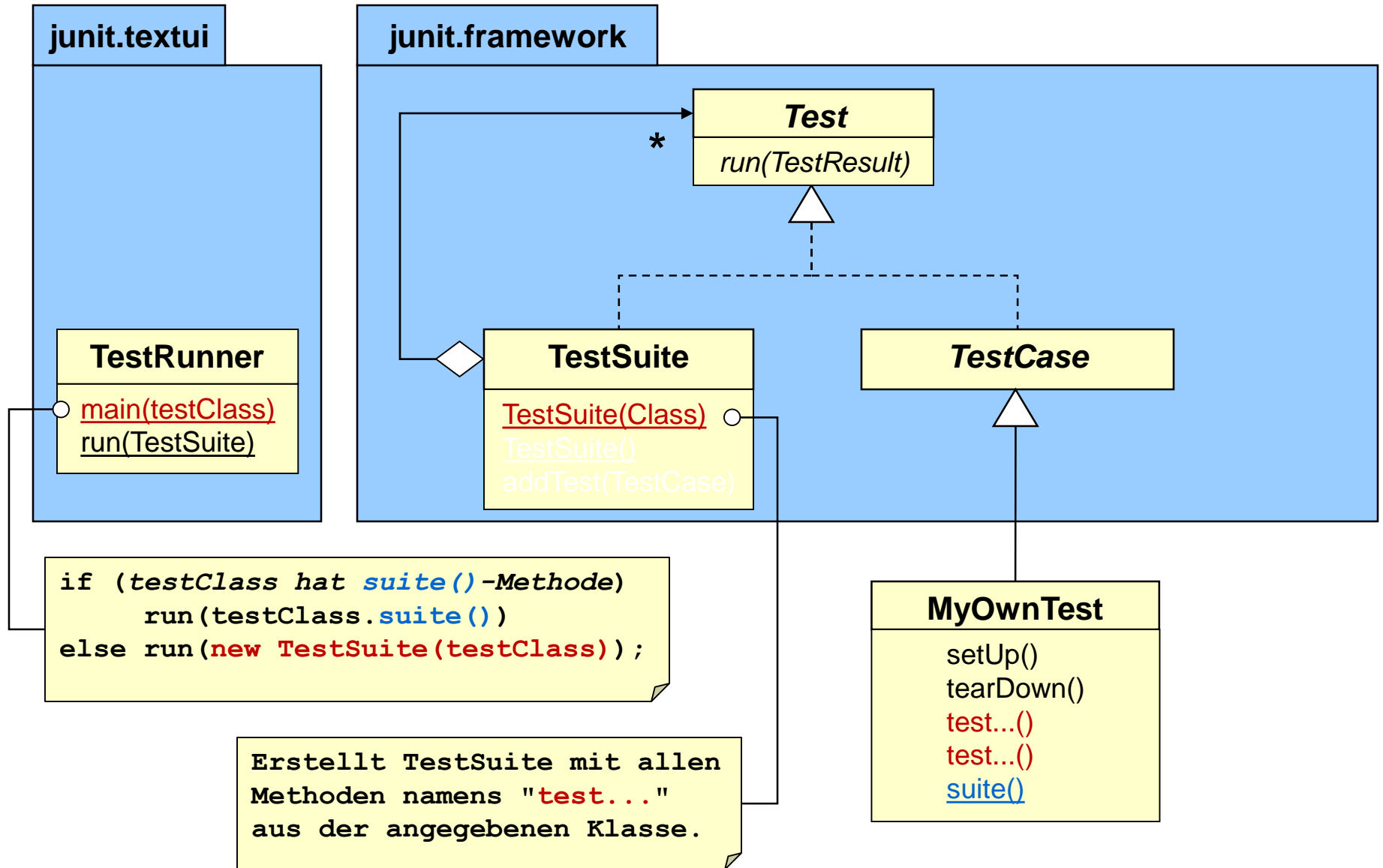
Der Kern von JUnit 3.x



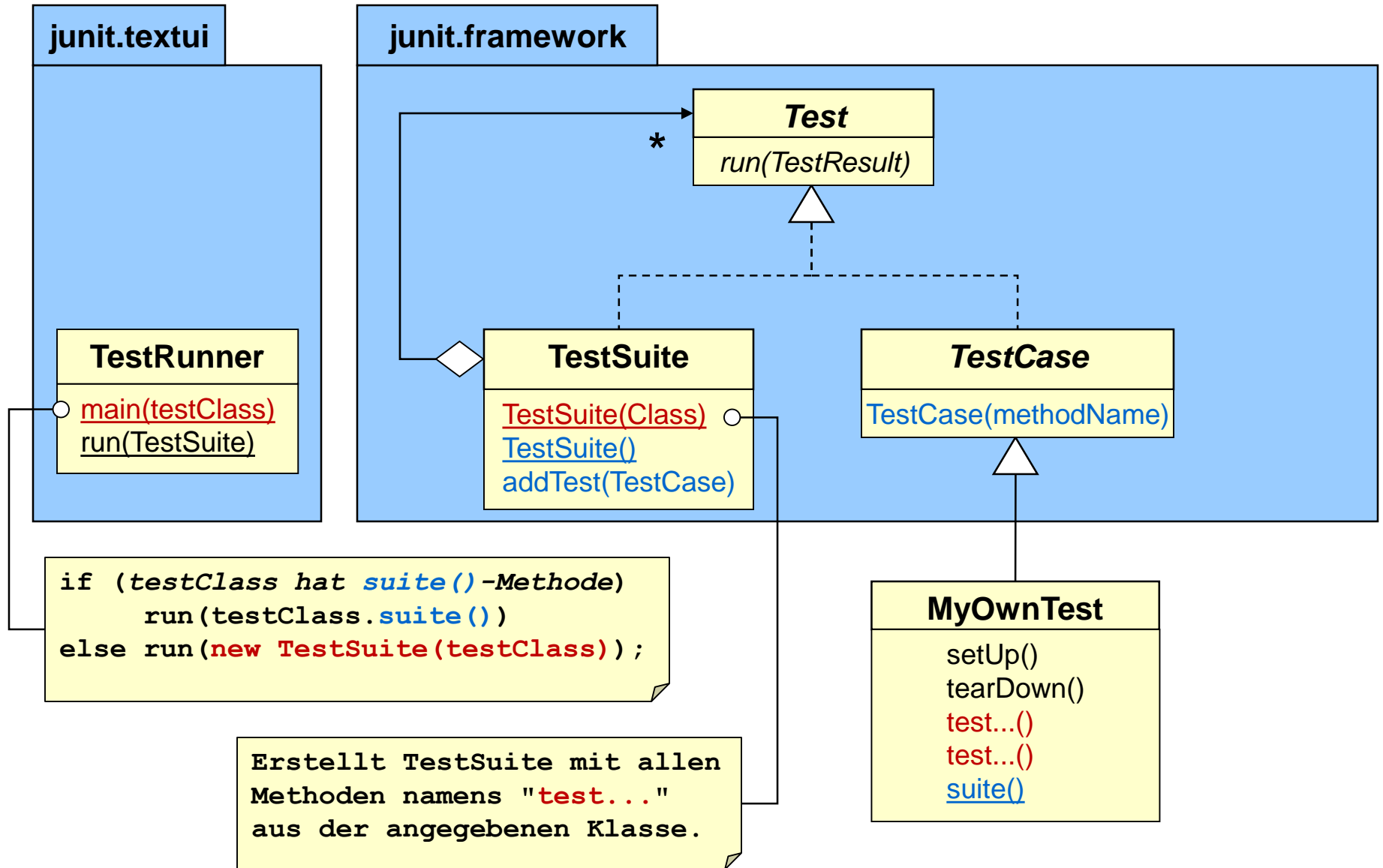
Start eines Testlaufs:

```
java junit.textui.TestRunner MyOwnTest
```

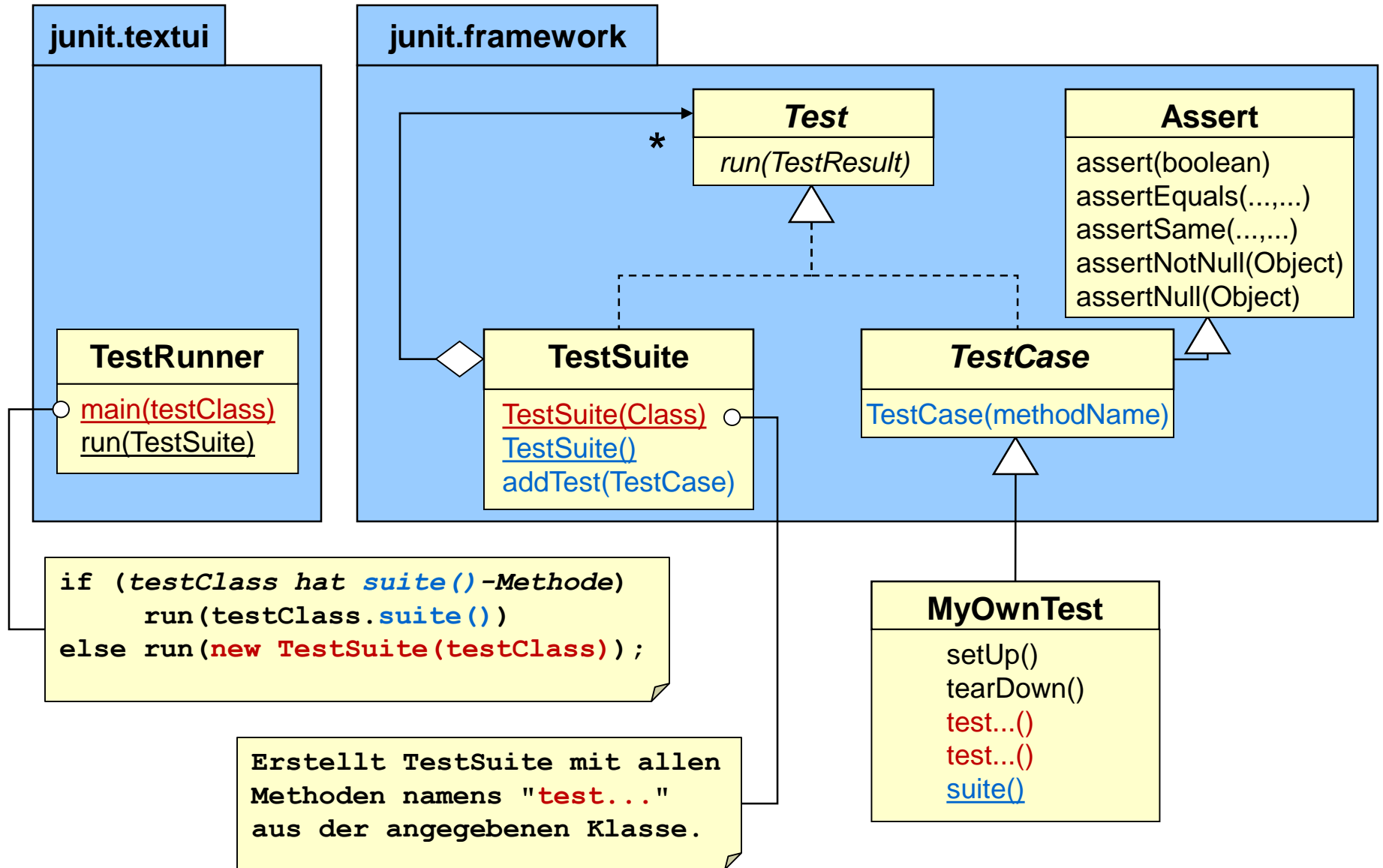
Der Kern von JUnit



Der Kern von JUnit



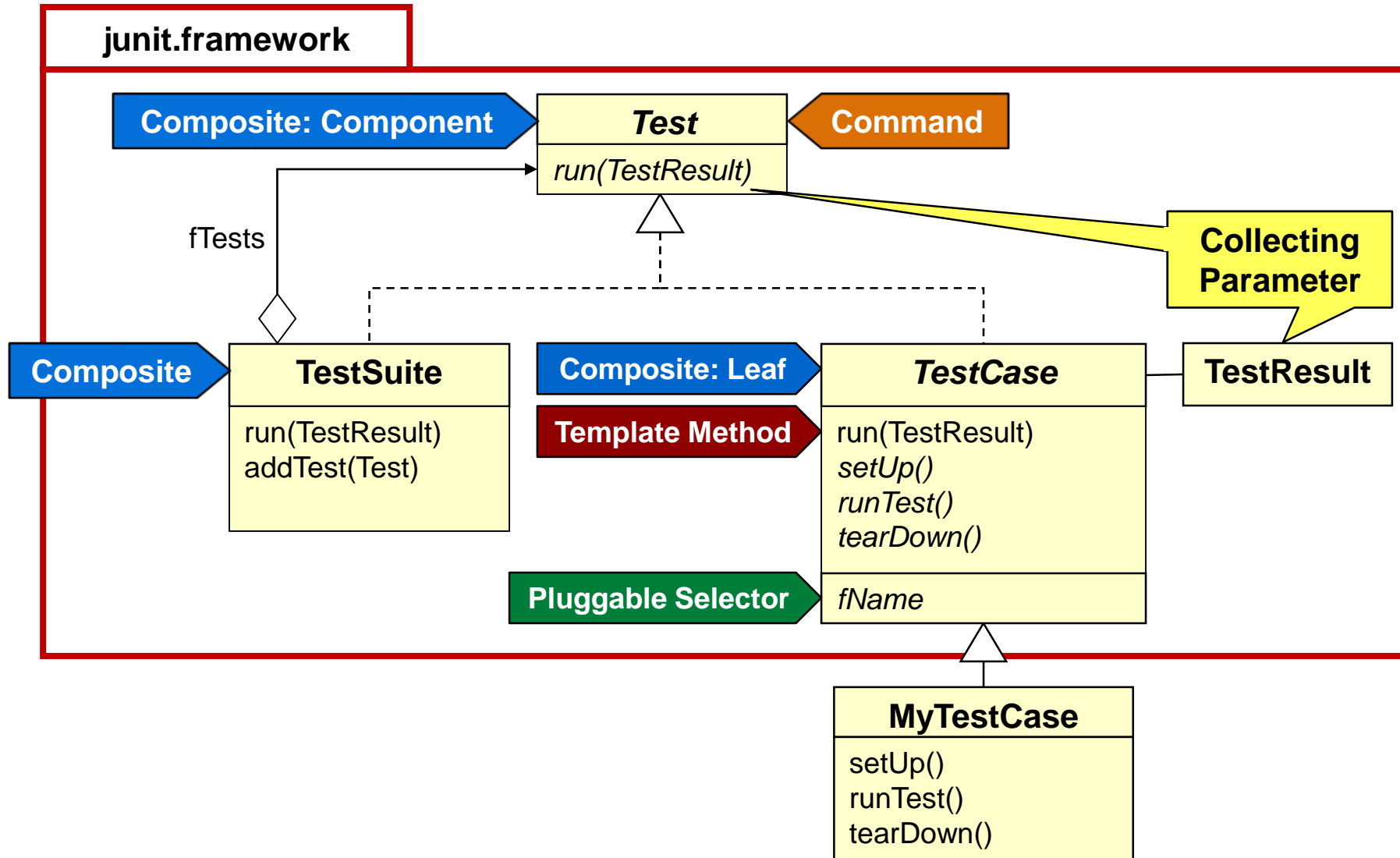
Der Kern von JUnit



Assertions

- Definition
 - ◆ Ausdrücke, die immer wahr sein müssen
 - ◆ Wenn nicht, meldet das Framework einen Fehler im aktuellen Test
 - ◆ ... und macht mit dem nächsten Test der Suite weiter
- Varianten
 - ◆ `assert(Boolean b)`
 - ⇒ minimalistische Fehlermeldung
 - ◆ `assertEquals(... expected, ... actual)`
 - ⇒ Gleichheit der Parameter hinsichtlich "equals()"-Methode
 - ⇒ viele überladene Varianten (double, long, Object, delta, message)
 - ◆ `assertSame(Object expected, Object actual)`
 - ⇒ Identität: Parameter verweisen auf das selbe Objekt
 - ◆ `assertNull(Object arg)`
 - ⇒ arg muss null sein
 - ◆ `assertNotNull(Object arg)`
 - ⇒ arg darf nicht null sein
 - ◆ `fail()`
 - ⇒ schlägt immer fehl → Testen von Exceptions!
 - ◆ immer auch Varianten mit "String message" als zusätzlichem ersten Argument

Struktur des Framework: "Patterns Create Architectures"



Benutzung von JUnit (1)

- Test erstellen

- ◆ eigene Unterklasse von `TestCase` (z.B.: **MyOwnTest**)

- ◆ implementiere Methoden

- ⇒ `setUp()` Testdaten erzeugen

- ⇒ `test...()` Test durchführen

- ⇒ `tearDown()` Ressourcen freigeben

- Test durchführen

- ◆ Einfachste Variante: `java junit.textui.TestRunner MyOwnTest`

Benutzung von JUnit (2)

- Testklasse
 - ◆ fasst zusammengehörige Testdaten und Testfälle zusammen
- Test-Suites
 - ◆ automatisch aus Testklasse generiert
- Testfall
 - ◆ eine Methode
 - ◆ nutzt Assertions
- Assertions
 - ◆ automatische Überprüfung der Ergebnisse
 - ◆ automatische Fehlerprotokollierung
- Komponierbarkeit
 - ◆ Test reihenfolge-unabhängig komponierbar

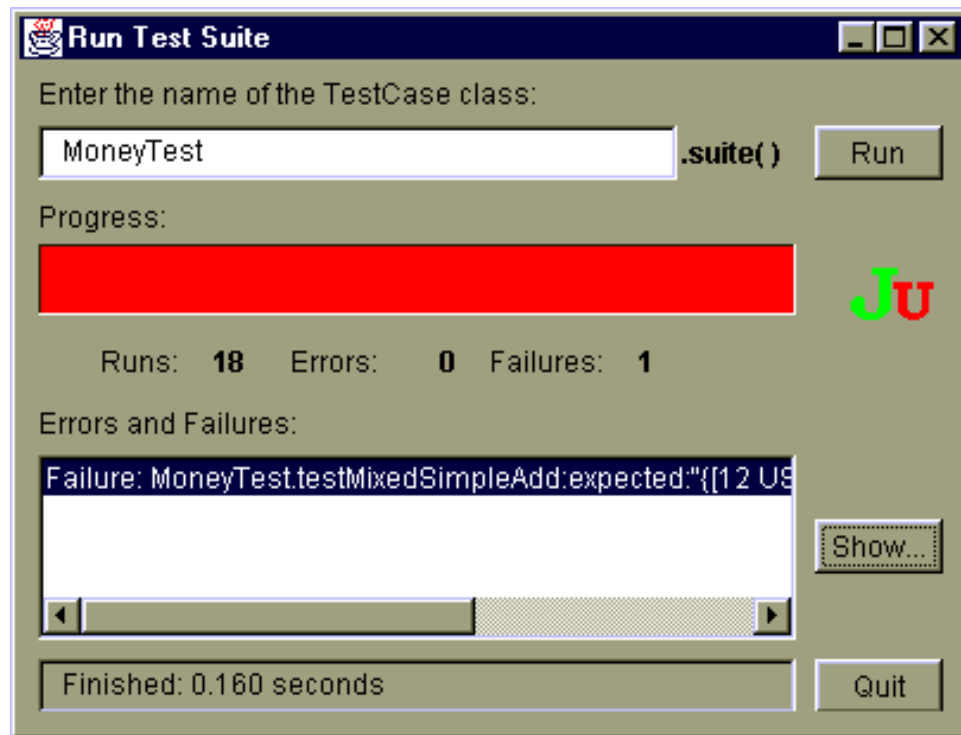
TestRunner-UI

- Kommandozeilen-Schnittstelle
 - ◆ junit.textui.TestRunner

```
.....  
OK!
```

```
.....F  
  
!!! FAILURES!!!  
Test Results:  
Run: 18 Failures: 1 Errors 0  
There was 1 failure:  
1) FileReaderTester.testRead  
   expected: "m" but was "d"
```

- Graphische Schnittstelle
 - ◆ junit.textui.TestRunner
 - ◆ junit.textui.LoadingTestRunner
 - ➔ lädt geänderte Klassen automatisch nach



- Refactoring und Tests

- Das Junit-Framework

- ◆ Einführung

- Beispiel

- ◆ Testen von GUIs

- Empfehlungen

Besipiel: Testen der FileReader-Klasse des JDK

- Testplanung: Spezifikation der Klasse studieren → Testfälle ableiten
- Testdaten erstellen (Testdatei)
- Testklasse schreiben
- Testen
- Testklasse erweitern
- Testen

Beispiel: Testdaten erstellen

- Testdaten zum Lesen aus Datei
 - ◆ Testdatei mit bekanntem Inhalt
 - ◆ 182 Zeichen lang

Datei "data.txt"							
Bradman	99.94	52	80	10	6996	334	29
Pollock	60.97	23	41	4	2256	274	7
Headley	60.83	22	40	4	2256	270*	10
Sutcliffe	60.73	54	84	9	4555	194	16

- Einbinden der Testdatei in setUp()
- Schliessen der Testdatei in tearDown()

Besipiel: Testen der FileReader-Klasse

```
class FileReaderTester extends TestCase {  
  
    public FileReaderTester(String methodName) {  
        super(methodName);  
    }  
  
}
```

```
private FileReader _input;  
  
protected void setUp() {  
    try {  
        _input = new FileReader("data.txt");  
    } catch (FileNotFoundException e) {  
        throw new RuntimeException (e.toString());  
    }  
}
```

```
protected void tearDown() {  
    try {  
        _input.close();  
    } catch (IOException e) {  
        throw new RuntimeException ("error on closing test file");  
    }  
}
```

```
public void testRead() throws IOException {  
    char ch = '&';  
    for (int i=0; i<4; i++)  
        ch = (char) _input.read();  
    assert('d' == ch);  
}
```

Datei "data.txt"

Bradman	99.94	52	80	10	6996	334	29
Pollock	60.97	23	41	4	2256	274	7
Headley	60.83	22	40	4	2256	270*	10
Sutcliffe	60.73	54	84	9	4555	194	16

Beispiel: Testfall erweitern



Grenzbedingungen testen!

- ◆ erstes Zeichen
- ◆ letztes Zeichen
- ◆ "endOfFile"
- ◆ nach "endOfFile"

Datei "data.txt"							
Bradman	99.94	52	80	10	6996	334	29
Pollock	60.97	23	41	4	2256	274	7
Headley	60.83	22	40	4	2256	270*	10
Sutcliffe	60.73	54	84	9	4555	194	16

```
class FileReaderTester extends TestCase {

    public FileReaderTester(String methodName) {
        super(methodName);
    }

    ...

    private final int _fileLength = 182;
    private final int _endOfFile = -1;

    public void testReadBoundaries() throws IOException {
        assertEquals("read first char", 'B', _input.read());
        int ch;
        for (int i=1; i<_fileLength-1; i++)
            ch = _input.read();
        assertEquals("read last char", '6', _input.read());
        assertEquals("read at end", _endOfFile, _input.read());
        assertEquals("read past end", _endOfFile, _input.read());
    }
}
```

Beispiel: Testfall erweitern (2)



Grenzbedingungen testen!

◆ leere Datei

```
// Erweitertes Testdaten-Setup:

private File _empty;

protected void setUp() { // overrides
    try {
        _input = new FileReader("data.txt");
        _empty = newEmptyFile(); // <-- added
    } catch (FileNotFoundException e) {
        throw new RuntimeException (e.toString());
    }
}

private FileReader newEmptyFile() throws IOException {
    File empty = new File ("empty.txt");
    FileOutputStream out = new FileOutputStream(empty);
    out.close();
    return newFileReader (empty);
}

// Added Tests:

public void testEmptyRead() throws IOException {
    assertEquals(_endOfFile, _empty.read());
}
```

Beispiel: Testfall erweitern (3)



Fehlerfälle testen!

- ◆ Testen ob beim Lesen aus einer nicht geöffneten Datei die erwartete Exception geworfen wird.

```
// Added Tests:  
  
public void testReadAfterClose() throws IOException {  
    _input.close(); // provoziere Fehler!  
    try {  
        _input.read(); // Leseversuch  
        fail("no exception for read past end"); // Hierher sollten wir nicht gelangen  
    } catch (IOException io) {}  
}
```

- ◆ Prinzip: fail()-Assertion markiert die Stelle die nicht erreicht werden dürfte, falls die getestete Methode die erwartete Exception wirft.

Beispiel: Explizite Auswahl der Testfälle

- Test-Suite-Erstellung

- ◆ automatisch

- ◆ explizit

```
// Test mit Default-Test-Suite:
```

```
public static void main (String[] args) {  
    junit.textui.TestRunner.run (new TestSuite(FileReaderTester.class));  
}
```

```
// Test mit selbst angepasster Test-Suite:
```

```
public static void main(String[] args) {  
    junit.textui.TestRunner.run (suite());  
}
```

```
// Explizite Test-Suite-Erstellung:
```

```
public static Test suite() {  
    TestSuite suite = new TestSuite();  
    suite.addTest(new FileReaderTester("testRead"));  
    suite.addTest(new FileReaderTester("testReadAtEnd"));  
    suite.addTest(new FileReaderTester("testReadBoundaries"));  
    return suite;  
}
```

JUnit 4

- Gleiche Grundideen aber leichter zu implementieren
 - ◆ Testklassen müssen nicht mehr von TestCase abgeleitet werden
 - ◆ Markierung von Testklassen und Methoden durch Java-Annotationen
- Annotationen
 - ◆ `@Test` → Testmethode
 - ◆ `@Before`
 - ◆ `@After`
 - ◆ `@BeforeClass`
 - ◆ `@AfterClass`

```
@Test
public void addition() {
    assertEquals(12, simpleMath.add(7, 5));
}
@Test
public void subtraction() {
    assertEquals(9, simpleMath.subtract(12, 3));
}
@Before
public void runBeforeEveryTest() {
    simpleMath = new SimpleMath();
}
@After
public void runAfterEveryTest() {
    simpleMath = null;
}
@BeforeClass
public static void runBeforeClass() {
    // run for one time before all test cases
}
@AfterClass
public static void runAfterClass() {
    // run for one time after all test cases
}
```

JUnit 4 (Fortsetzung)

- Exceptions

- ◆ Parameter „expected“
- ◆ Angabe der Exception-Klasse

```
@Test(expected = ArithmeticException.class)
public void divisionWithException() {
    simpleMath.divide(1, 0); // divide by zero
}
```

- Timeouts

- ◆ Parameter „timeout“
- ◆ Angabe der Millisekunden

```
@Test(timeout = 1000)
public void infinity() {
    while (true);
}
```

- Deaktivierte Tests

- ◆ Annotation „ignore“
- ◆ Erläuterung als Parameter

```
@Ignore("Not Ready to Run")
@Test
public void multiplication() {
    assertEquals(15, simpleMath.multiply(3, 5));
}
```

Kurze Demo von JUnit 3 und JUnit 4: Kreditverlaufsrechnung

Berechnungsergebnis

Ihre Angaben waren:

- Kredithöhe : 100.000,00 EURO
- Nominalzins : 5,00 Prozent pro Jahr
- Laufzeit : 5 Jahr(e)
- Anzeige : mit Monatsraten

Ihre Ergebnisse sind:

- Gesamtaufwand : 113.227,40 EURO
- davon Zinsen : 13.227,40 EURO
- davon Tilgung : 100.000,00 EURO

Tilgungsplan Annuitätendarlehen

(M)	Rate	Zins	Tilgung	Restschuld
1	1.887,12	416,67	1.470,46	98.529,54
2	1.887,12	410,54	1.476,58	97.052,96
3	1.887,12	404,39	1.482,74	95.570,22
4	1.887,12	398,21	1.488,91	94.081,31
5	1.887,12	392,01	1.495,12	92.586,19
6	1.887,12	385,78	1.501,35	91.084,84
7	1.887,12	379,52	1.507,60	89.577,24
8	1.887,12	373,24	1.513,88	88.063,36
9	1.887,12	366,93	1.520,19	86.543,16
10	1.887,12	360,60	1.526,53	85.016,64
11	1.887,12	354,24	1.532,89	83.483,75
12	1.887,12	347,85	1.539,27	81.944,47

Test-First und Continuous Testing

Test-Driven Development
Continuous Testing

Wann soll man Modul-Tests schreiben?

- Wenn die Klasse fertig ist?
 - ◆ Testen bevor andere damit konfrontiert werden.
- Parallel zur Implementierung der Klasse
 - ◆ Testen um eigene Arbeit zu erleichtern.
- Vor der Implementierung der Klasse!
 - **TDD: Test-Driven Development!**
 - ◆ Konzentration auf Interface statt Implementierung
 - ◆ Durch Nachdenken über Testfälle Design-Fehler finden bevor man sie implementiert!
 - ◆ Tests während Implementierung immer verfügbar
 - ⇒ Laufendes Feedback und Erfolgskontrolle

„Continuous Testing“

- Beobachtung
 - ◆ Je kürzer das Intervall zwischen Änderung und Test ist, um so schneller ist die Fehlerquelle lokalisierbar
- Idee
 - ◆ Tool lässt Tests nach jeder Änderung im Hintergrund laufen
 - ◆ Programmierer konzentriert sich voll auf Entwicklungsaufgaben
 - ⇒ muss nur noch auf Testfehlschläge reagieren, nicht mehr selbst testen
 - ◆ Unaufdringliches Feedback
 - ⇒ Fehlerübersicht im „Problem View“
 - ⇒ Fehlermarker in fehlgeschlagenen Tests
 - ⇒ junit-Fenster nur bei Bedarf (Stack-Trace-Anzeige)
- Continuous Testing Infos und Tools
 - ◆ <http://groups.csail.mit.edu/pag/continouostesting/>
 - ◆ „Infinittest“ (GPL Lizenz): <http://infinittest.org/>

„Continuous Testing“

► Infinitest

```
public void testTopThreeIntsUnsorted() {  
    Integer one = new Integer(1);  
    Integer two = new Integer(2);  
    Integer three = new Integer(3);  
    Integer four = new Integer(4);  
    Integer five = new Integer(5);  
}
```

Doppel-Klick

Description	Resource	In Folder
Test failure: testTopThreeIntsUnsorted(TopTenTest)	TopTenTest.java	topten/src

- Go To
- Show in Navigator
- Copy
- Paste
- Delete
- Select All
- Quick Fix
- Rerun Tests
- Delete All Failures
- Delete Marker
- View Trace**
- Properties

Problems Javadoc Declaration Console JUnit

Finished after 0.031 seconds

Result:

CT Launch For Project topten

Runs: 4/4 Errors: 0 Failures: 1

Failures Hierarchy

testTopThreeIntsUnsorted - topten.TopTenTest

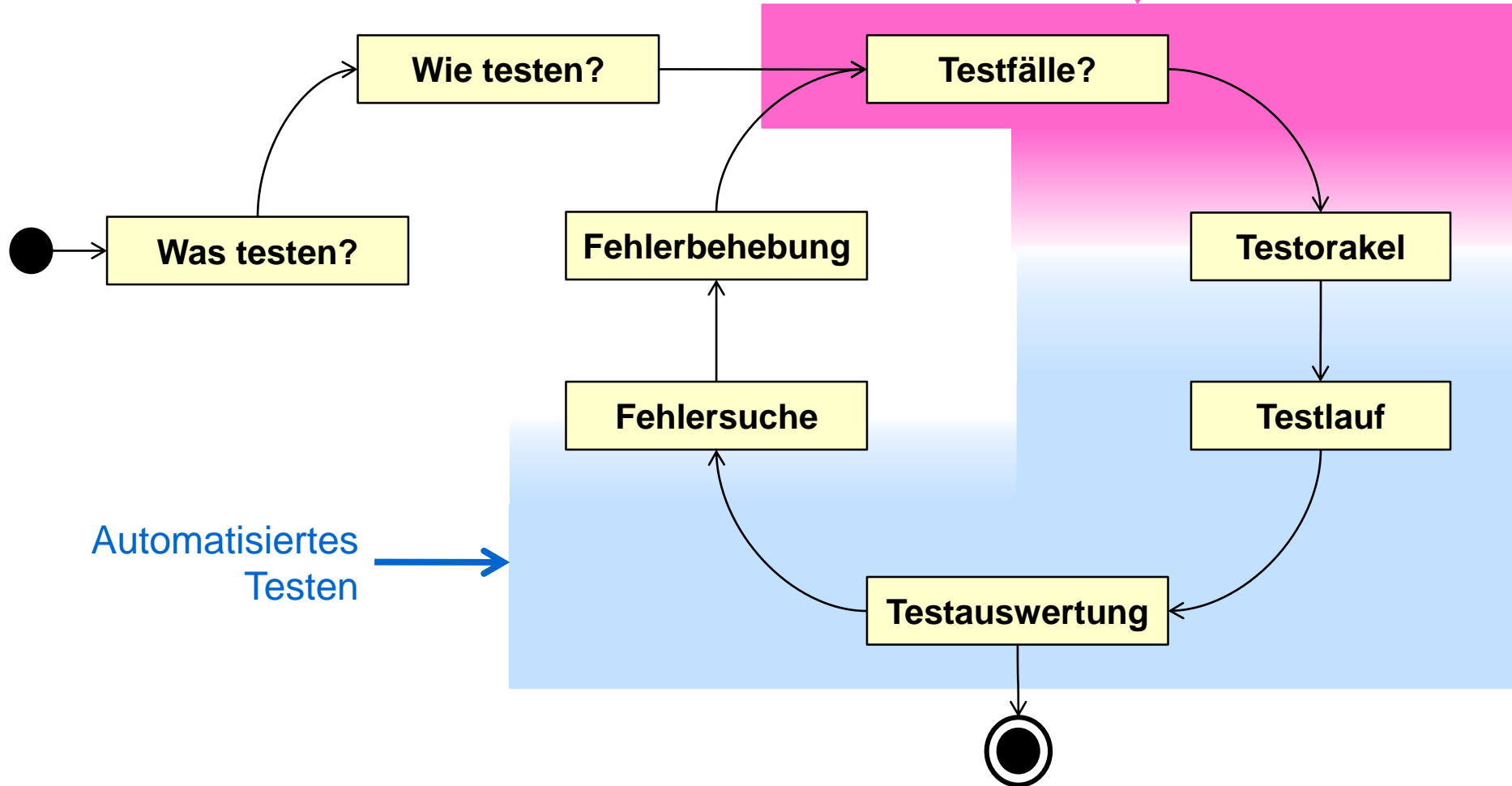
Failure Trace

```
junit.framework.AssertionFailedError: expected: <[1, 2, 3]> but  
at topten.TopTenTest.testTopThreeIntsUnsorted(TopTenTest.ja  
at sun.reflect.NativeMethodAccessorImpl.invoke0(Native Metho  
at sun.reflect.NativeMethodAccessorImpl.invoke(Unknown Sourc
```


Automatisierte Testgenerierung

Automatisierte Testerzeugung

Automatisierte
Testerzeugung



Automatisierte Testgenerierung mit T3

- Idee

- ◆ Schreiben von Code laut Design-by-Contract
- ◆ Testfälle (= Aufrufsequenzen) werden generiert!
- ◆ Überprüfung von Vorbedingungen, Nachbedingungen und Invarianten ersetzt Orakel

- Umsetzung

- ◆ T3 framework: <http://code.google.com/p/t2framework/>

- DBC in T3

- ◆ Vorbedingungen durch Java Assertions mit postfix : "PRE" spezifizieren
 - ⇒ `assert !s.isEmpty() : "PRE" ;`
- ◆ Nachbedingung durch Java Assertions mit postfix : „POST" spezifizieren
 - ⇒ `assert s.isEmpty() : "POST" ;`
- ◆ Klasseninvarianten durch spezielle Methode spezifizieren
 - ⇒ `private boolean classinv__() { return s.isEmpty() || s.contains(max) ; }`

Automatisierte Testgenerierung mit T3

- Testfallgenerierung
 - ◆ Beliebige Aufrufsequenzen von Methoden aus der zu testenden Klasse
 - ◆ Hohe Anzahl an Sequenzen (Hunderte)
- Testlauf
 - ◆ Sequenzen ausführen, inklusive Assertion-Auswertung
 - ⇒ Invarianten sowohl vor als auch nach jedem Aufruf
- Bericht
 - ◆ Gesamtzahl der Sequenzen
 - ◆ Sequenzen die durchlaufen, für die aber Assertions fehlschlagen
 - ◆ Sequenzen die Exceptions werfen (\neq AssertionException)

T3-Beispiel: Sortierte Integer-Liste

```
public class SortedIntegerList {  
  
    private LinkedList<Integer> s ;  
    private Integer max ;  
  
    public SortedList() { s = new LinkedList<Integer>() ; }  
  
    public void insert(Integer x) {  
  
        int i = 0 ;  
        for (Integer y : s) {  
            if (y >x) break ;  
            i++ ;  
        }  
        s.add(i,x) ;  
        if (max==null || x < max) max = x ; // ← Fehler: Sollte x > max sein!  
    }  
  
    public Integer get() {  
  
        Integer x = max ;  
        s.remove(max) ;  
        if (s.isEmpty()) max = null ;  
        else max = s.getLast() ;  
  
        return x ;  
    }  
}
```

T3-Beispiel: Spezifikationen im Code

```
public class SortedIntegerList {

    private LinkedList<Integer> s ;
    private Integer max ;

    public SortedList() { s = new LinkedList<Integer>(); }

    private boolean classinv__() { return s.isEmpty() || s.contains(max) ; }           // Invariant

    public void insert(Integer x) {
        assert x!=null : "PRE";
        int i = 0 ;
        for (Integer y : s) {
            if (y >x) break ;
            i++ ;
        }
        s.add(i,x) ;
        if (max==null || x < max) max = x ; // ← Fehler: Sollte x > max sein!
    }

    public Integer get() {
        assert !s.isEmpty() : "PRE" ;                                               // Pre-condition
        Integer x = max ;
        s.remove(max) ;
        if (s.isEmpty()) max = null ;
        else max = s.getLast() ;
        assert s.isEmpty() || x >= s.getLast() : "POST";                             // Post-condition
        return x ;
    }
}
```

Testen mit T3

- Aufruf von T3

```
java -ea -cp T3.jar Sequenic.T3.T3Cmd --showexc Examples.SimpleIntSortedList
```

enable
assertions

use
T3

show first failed
execution sequence

class to
be tested

- Ergebnis

```
Suite size : 36 -- T3 generated a set of 302 test-sequences
Average length : 8.972222 -- the average length of sequences,
                          in number of steps
Violating : 10 -- error found
Invalid : 1 -- number of invalid test-sequences
Failing : 0 -- number of failing/broken test-sequences
Runtime : 9 ms
```

```
... erste Sequenz die zu einer Assertion-Verletzung führte (aus Platzgründen
weggelassen - siehe http://uprime815.bitbucket.org/t3/docs/manual.html#toc0)
...
```

Unit Testing – Goldene Regeln

Unit Tests: Empfehlungen



Automatisierung

- ◆ Stelle sicher, daß alle Tests automatisiert ablaufen und ihre eigenen Ergebnisse überprüfen.



Ausdauer

- ◆ Führe Deine Tests regelmäßig durch.
- ◆ Teste nach jedem Kompilieren - mindestens einmal täglich.



Zuerst testen, dann debuggen

- ◆ Erhältst Du einen Fehlerbericht, schreibe erst einen Test, der den Fehler sichtbar macht.



Grenzbedingungen testen

- ◆ Konzentriere Deine Tests auf Grenzbedingungen, wo sich Fehler leicht einschleichen können.



Fehlerbehandlung testen

- ◆ Vergiss nicht zu testen, ob im Fehlerfall eine Exception ausgelöst wird.



Kein Perfektionismus

- ◆ Lieber unvollständige Test benutzen, als vollständige Tests nicht fertig bekommen.

Unit Tests: Empfehlungen



„Es gibt nichts Gutes, außer man tut es!“ (Erich Kästner)

- ◆ Tests können **nicht alle** Fehler finden.
- ◆ Lassen Sie sich davon nicht abhalten die paar Tests zu schreiben, die bereits **die meisten** Fehler finden!

Testen von GUIs

GUI (Graphical User Interface)

- Hierarchie von “Widgets”
 - ◆ Widget W_i = graphisches Objekts mit einer Menge von
 - ⇒ Eigenschaften (‘properties’ p_{ij}) mit jeweils eigenem
 - ⇒ Diskretem Wert (value v_{ij}) zur Laufzeit
- GUI Zustand (State) $S_t = \{ \dots, (w_i, p_{ij}, v_{ij}), \dots \}$
 - ◆ Werte aller Eigenschaften aller Widgets zum Zeitpunkt t
- GUI Ereignis (Event) E
 - ◆ Initiatiert transition aus Zustand S zu Zustand S' .

GUI-Testing Herausforderungen

- Protokollierung
 - ◆ GUI-Zustände erfassen
 - ◆ GUI-Übergänge verfolgen
- Automatisiertes Testen
 - ◆ Problem mit Zustandsexplosion
 - ◆ Explosion von Eventsequenzen, die den gleichen Effekt haben
- Technologabhängigkeit
 - ◆ Java AWT, Eclipse SWT, Web, Android, Win, MacOS, ...
- Automatisieren von GUI-Tests ist schwer
- Instandhaltung von GUI-Tests ist **sehr** schwer

GUI Testen ist schwer

I/O-Tests



GUI-Tests

- Methode Aufrufen
 - ◆ auf den Testobjekten
 - ◆ vom Test erstellt
- Input liefern
 - ◆ Argumente / globale Variablen
- Erwartungen prüfen
 - ◆ Rückgabewerte
- Technologieunabhängig
 - ◆ Methodenaufruf

- GUI-Event auslösen (z.B. Klick)
 - ◆ auf existierenden GUI-Komponenten
 - ◆ Komponenten müssen identifiziert werden!
- Input liefern
 - ◆ Textfelder füllen, ...
- Erwartungen prüfen
 - ◆ GUI-Struktur / -Aussehen
 - ◆ GUI-Verhalten
- Technologieabhängig
 - ◆ Java AWT, Eclipse SWT, Web, Android, Win, MacOs, ...

Testen von GUIs: Freie Tools

Jubula

- ◆ Von BREDEX GmbH Eclipse frei zur Verfügung gestellt
- ◆ Basis ihres kommerziellen Werkzeuges GUIDancer
- ◆ <http://eclipse.org/jubula/>

Zusammenfassung

- Automatisiertes Testen
 - ◆ Automatisiert Testlauf und Testauswertung
 - ◆ Ermöglicht Regressionstesting bei jeder Änderung
 - ◆ Continuous Testing → Inifinitest
- Test-Driven Development
 - ◆ Testfalldefinition bereits vor der Implementierung
 - ◆ Verbessertes Design und laufende Erfolgskontrolle
- JUnit
 - ◆ Assertion-Konzept für Spezifikation des Testorakels
 - ◆ Vereinfachte Testdefinition mit Annotationen (ab JUnit 4 / Java 5)
- Automatisierte Testgenerierung → T3
 - ◆ DBC-Spezifikation wird genutzt um Testfälle und Testorakel zu generieren