universitätbonn ROOTS

Kapitel 10b
Test-Automatisierung

Stand: 22.01.2014

Warum automatisierte Tests?

Automatisierte Modultests mit JUnit

‚Test First Development‘ und ‚Continuous Testing‘

Automatisierte Testerstellung mit T2

Test-Arten

- Modul-Test (Unit Test)
  - vor "check in" geänderter sourcen 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

Fokus im Folgenden: regressive Modul-Tests

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

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 wiederspiegeln, 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

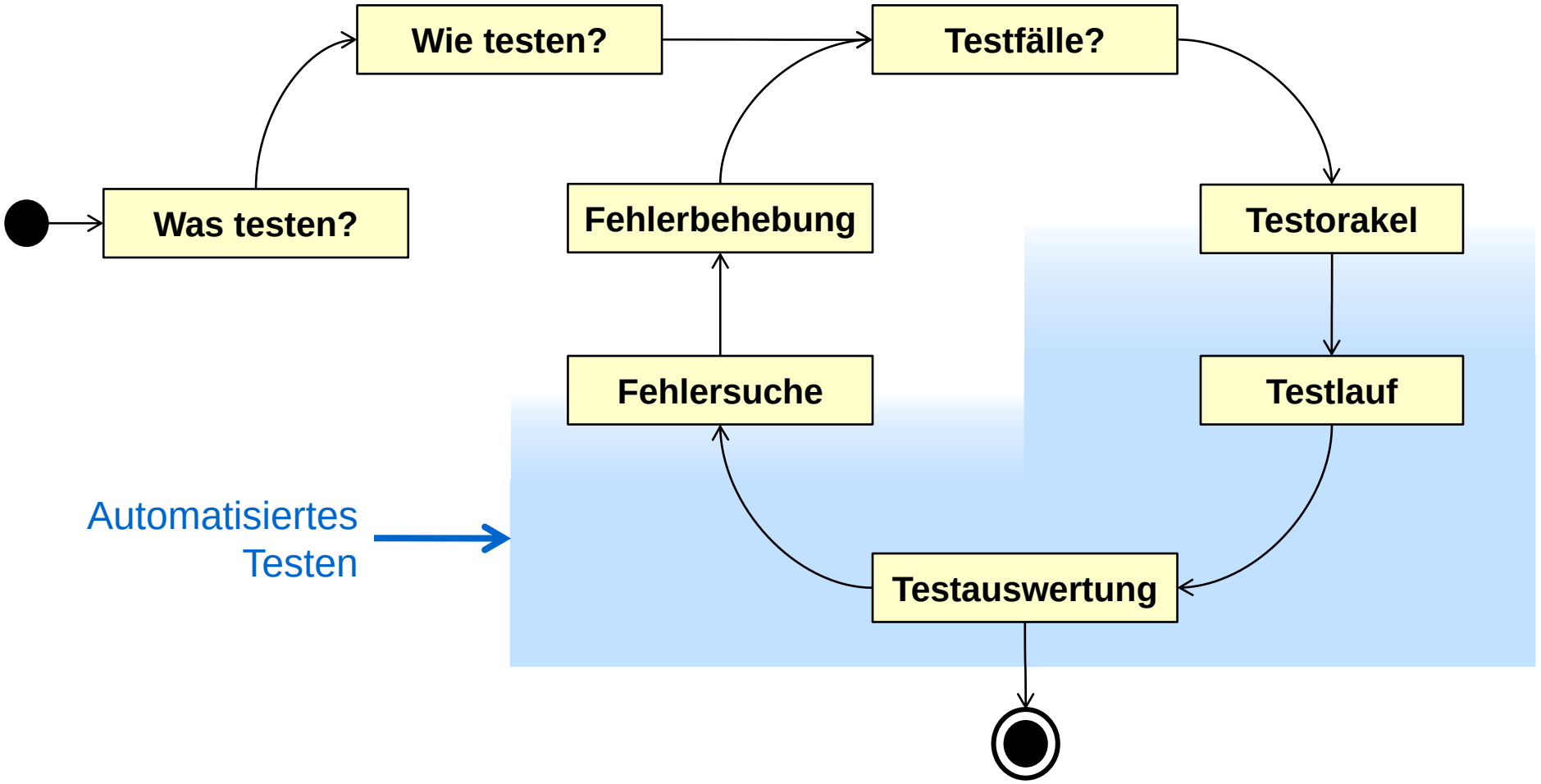
- Geringerer Aufwand
  - Tests zu schreiben
  - Tests zu warten
  - Tests zu aktivieren / deaktivieren
  - Tests zu komponieren
  - Tests durchzuführen und auszuwerten!!!!!!!!!!!!!!!!!!!!!!!!!!!!
- Testen in kürzeren Abständen möglich
  - Weniger Fehlerquellen zwischen Tests
  - Erinnerung was man verändert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

➔ Weniger Fehler

➔ Schnellere Identifikation der Fehlerursache

➔ Schnellere Programmentwicklung!!!

Automatisiertes Testen im Testzyklus



Tests

- Warum automatisierte Tests
- Das Junit-Framework
  - ➔ Einführung
  - ◆ Beispiel
  - ◆ Testen von GUIs
- Empfehlungen

The JUnit Test Framework

- Open source Framework
  - in Java, für Java
- Autoren
  - Kent Beck, Erich Gamma
- Web-Site
  - www.junit.org/

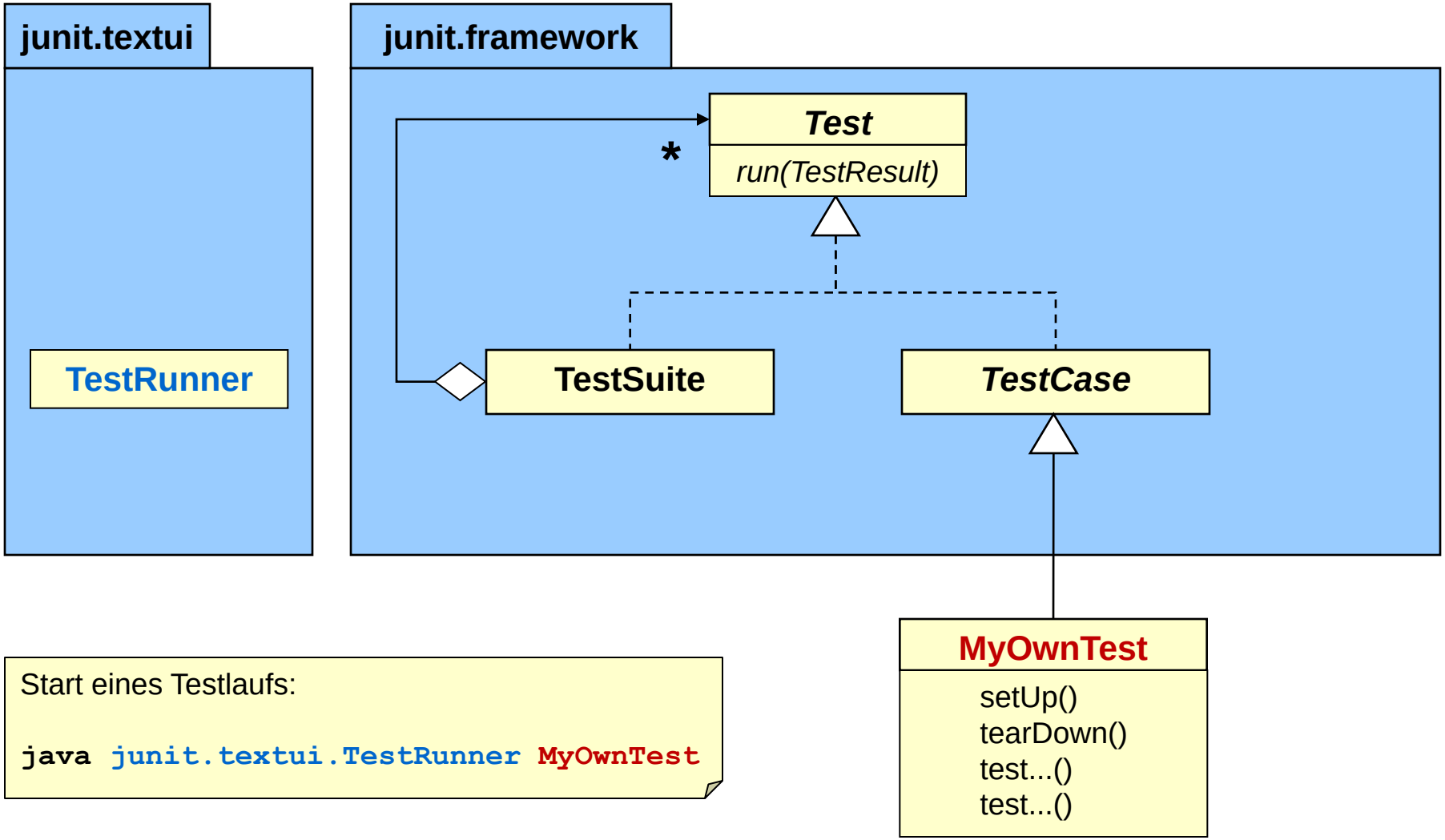
- Allgemeine Grundeinstellung
  - Der Programmierer meint „Das Feature funktioniert“⇒ Es funktioniert.
- JUnit Grundeinstellung
  - Es gibt keinen automatischen Test für das Feature⇒ Es funktioniert nicht.

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!

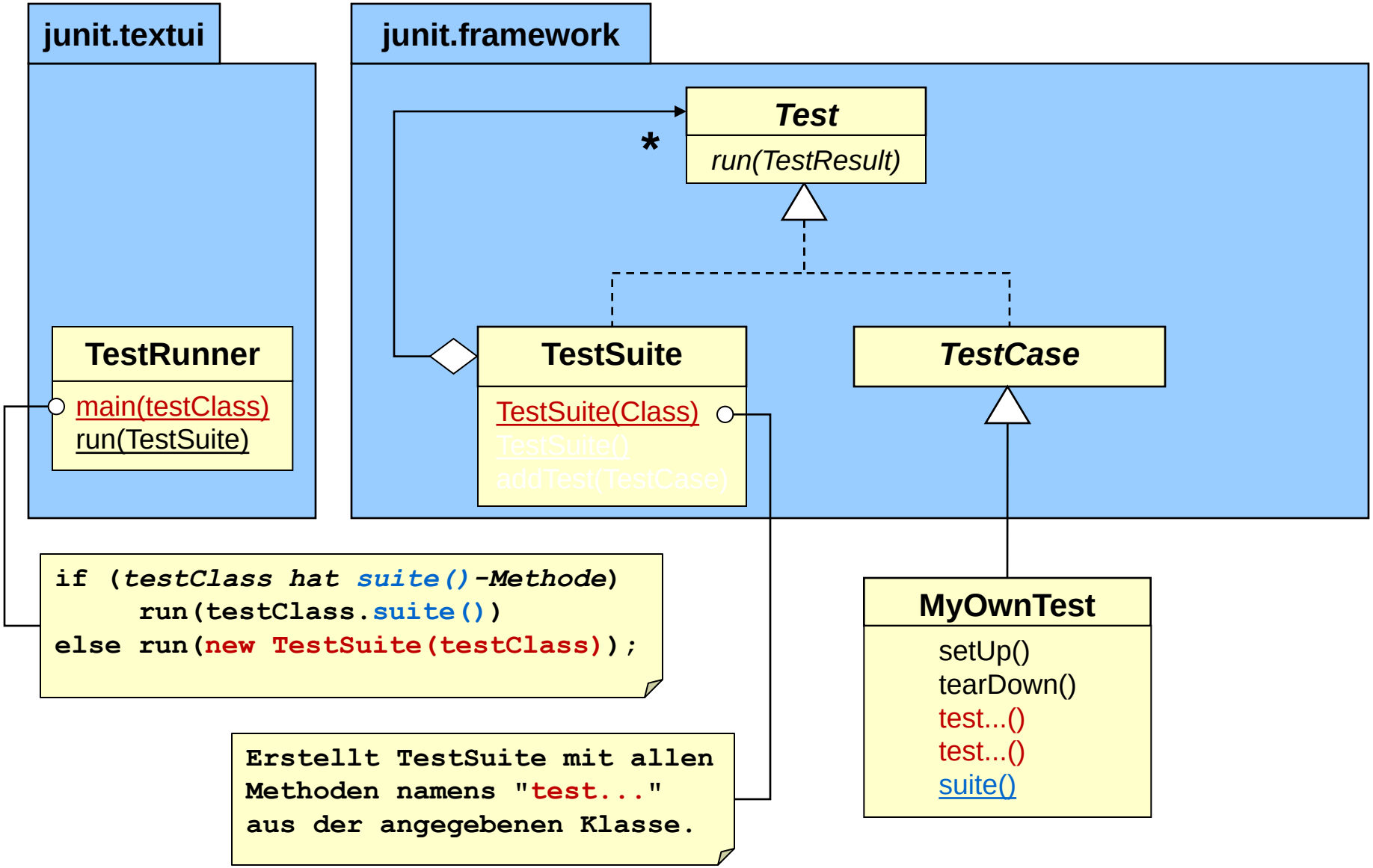
Der Kern von JUnit 3.x



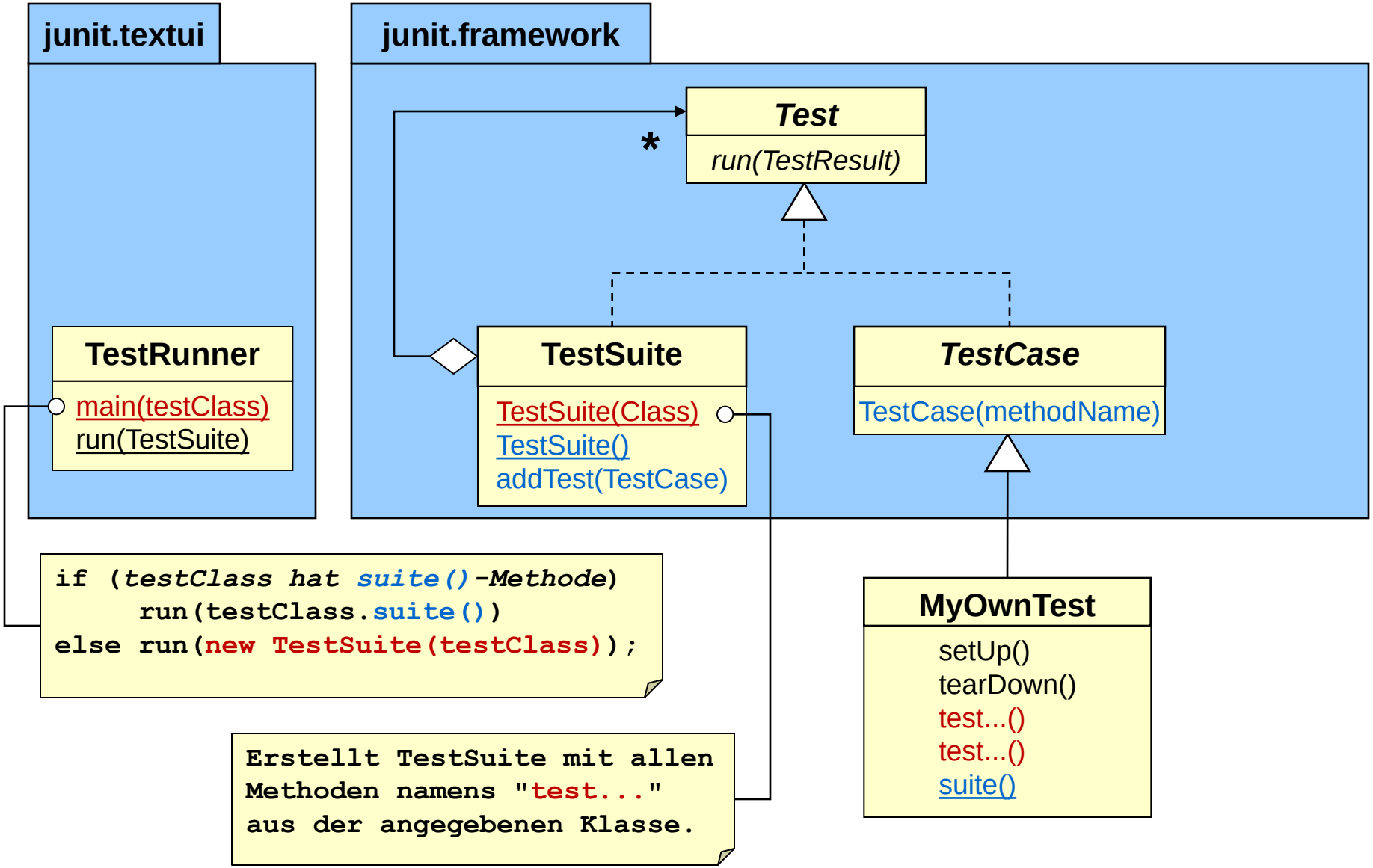
Start eines Testlaufs:

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

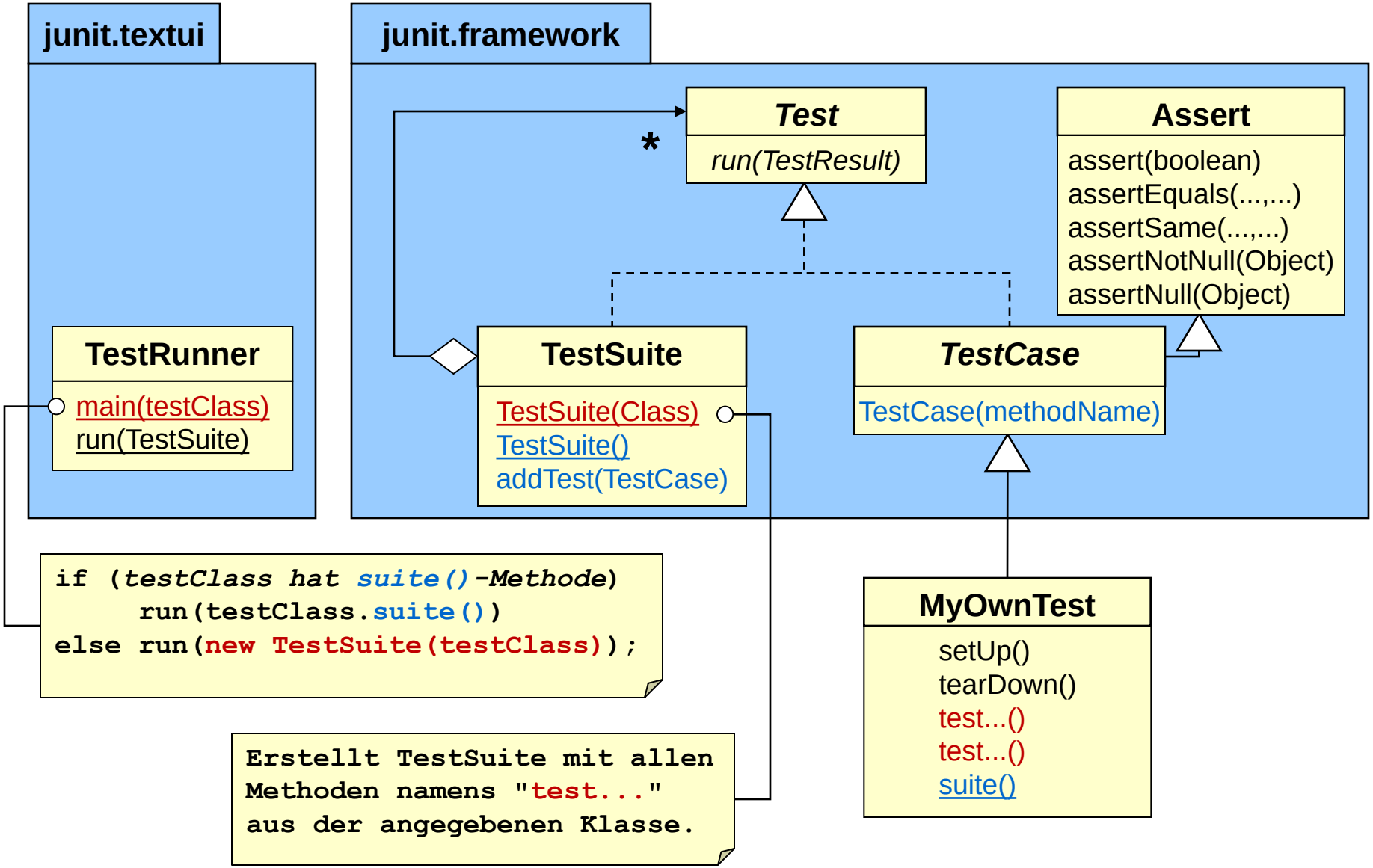
Der Kern von JUnit



Der Kern von JUnit



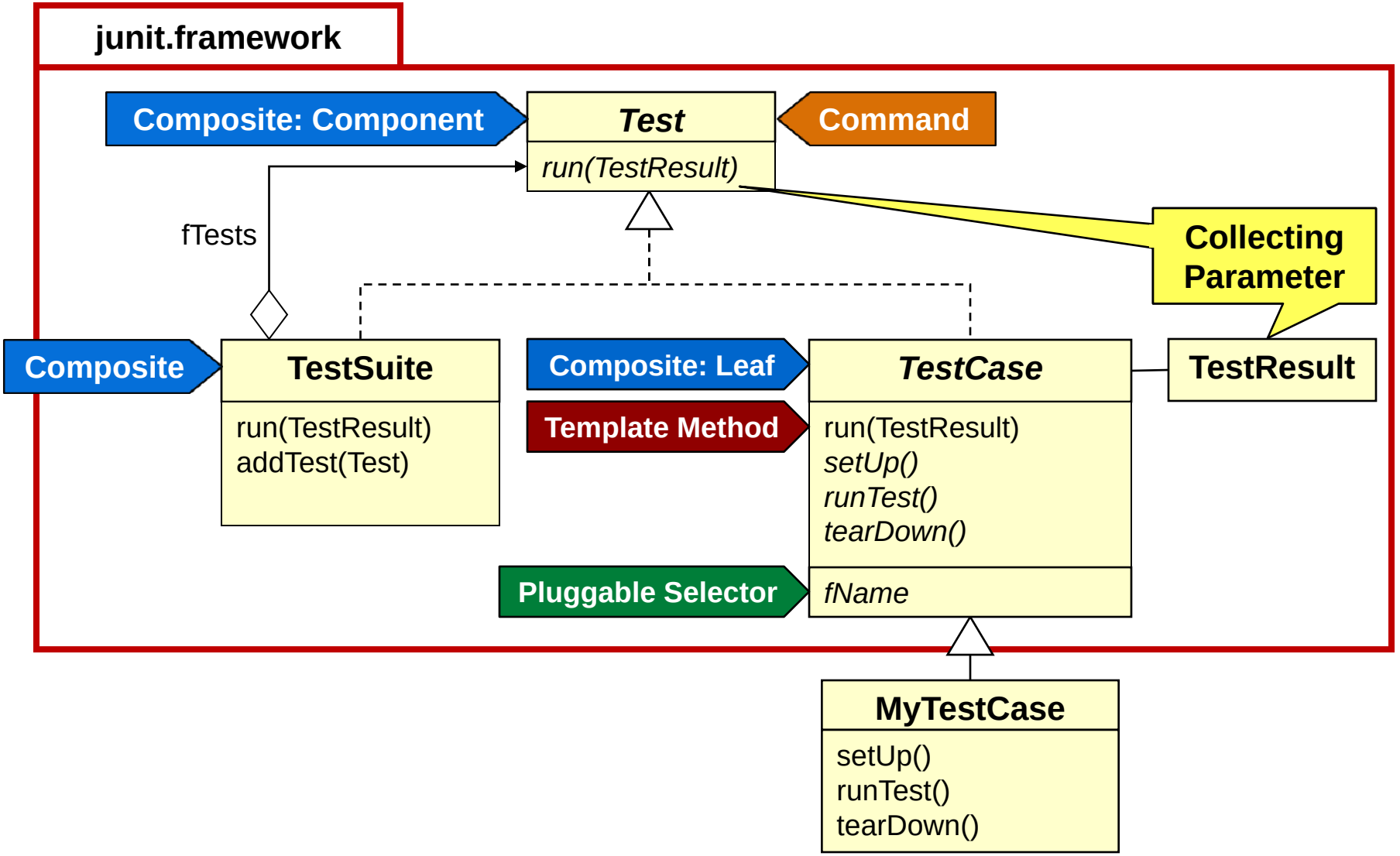
Der Kern von JUnit



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()` Resourcen 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

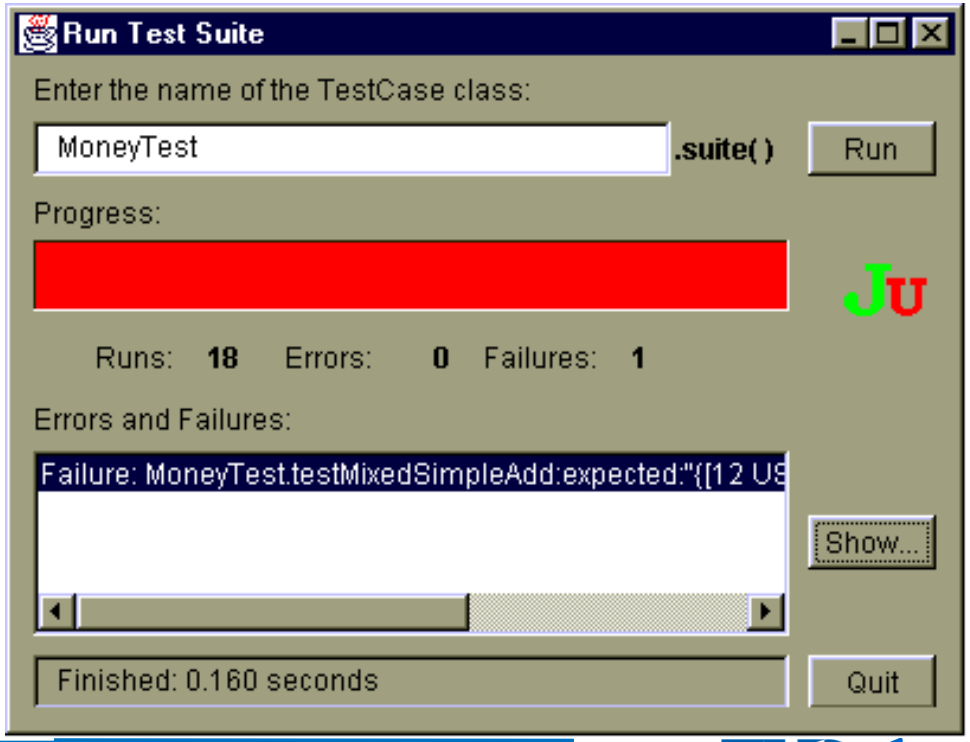
- Komandozeilen-Schnittstelle
  - junit.textui.TestRunner

```
.........................
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



Tests

- 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 = '&';
      vor (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.substract(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
  - Paremeter „expected“
  - Angabe der Exception-Klasse

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

- Timouts
  - Paremeter „timout“
  - 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 unjd JUnit 4: Kreditverlaufsberechnung

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

Unit Testing – Weiteres

Empfehlungen

Test-Driven Development

Continuous Testing

Automatisierte Testerzeugung

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

- Erhälst 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!

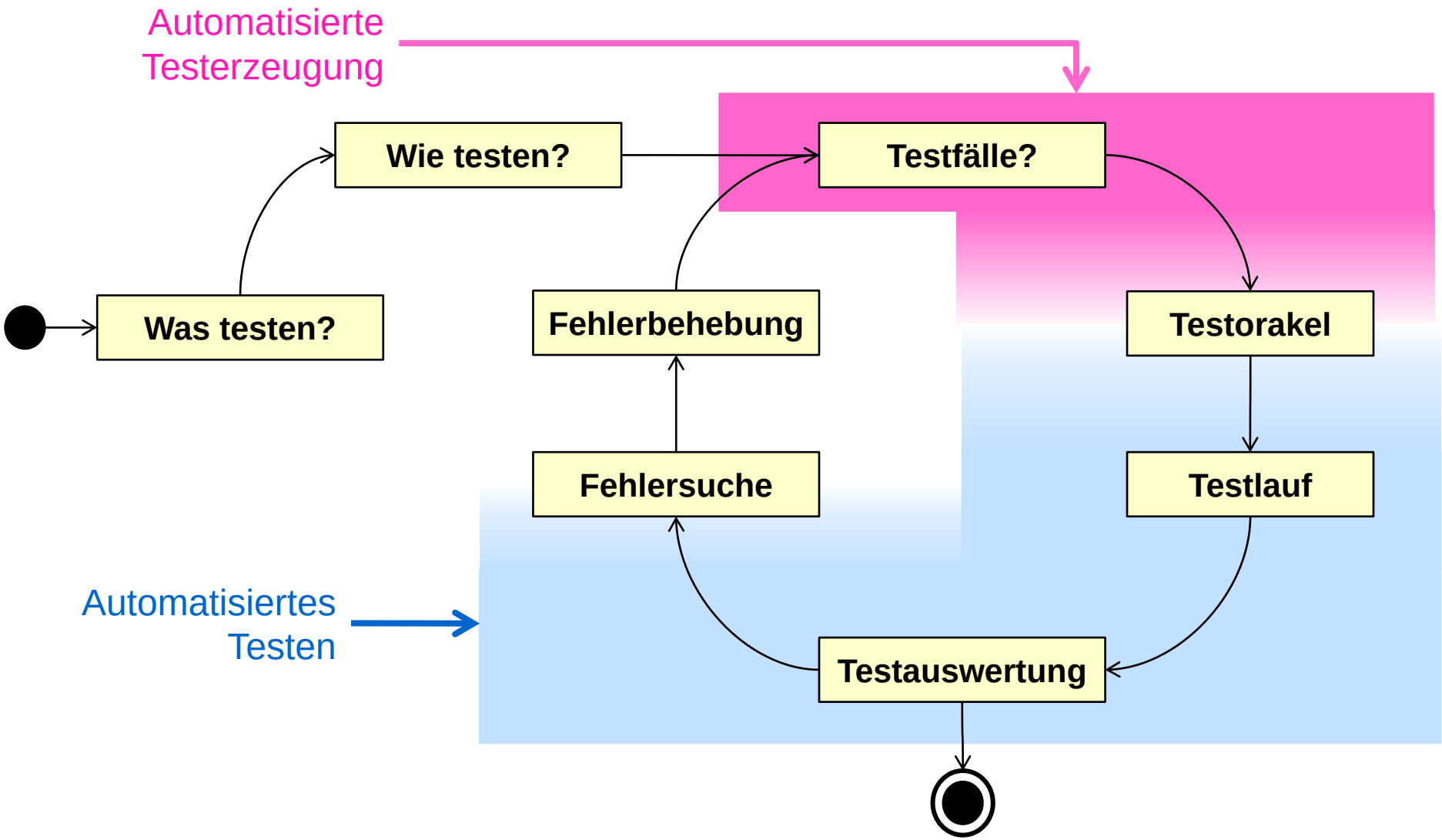
Wann soll man Modul-Tests schreiben?

- 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
  - Tests sind um so nützlicher, je kürzer das Intervall zwischen Änderung und Test ist
  - Die Fehlerquelle ist dadurch schneller lokalisierbar
- Idee: Laufend Testen!
  - Tool lässt Tests ständig im Hintergrund laufen
  - ... zeigt direkt den Code an, der von fehlgeschlagenen Tests betroffen ist
  - Programmierer konzentriert sich voll auf seine Entwicklungsaufgaben
  - ... muss nur noch auf Testfehlschläge reagieren, nicht mehr selbst testen
- Continuous Testing Infos und Tools
  - http://groups.csail.mit.edu/pag/continuoustesting/
  - http://blog.objectmentor.com/articles/2007/09/20/continuous-testing-explained
  - „Infinitest“-Tool für Java (GPL Lizenz): http://code.google.com/p/infinitest/

Automatisierte Testerzeugung



Automatisierte Testgenerierung mit T2

- Idee
  - Schreiben von Spezifikationen statt Testfällen und Orakeln
  - Testfälle und Orakel werden generiert!

- Vorgehen
  - Spezifikation ist Teil des Codes
  - Klasseninvarianten durch spezielle Methode spezifizieren
    - ⇨ `private boolean classinv() { return s.isEmpty() || s.contains(max) ; }`
  - Vorbedingungen durch Java Assertions mit postfix `: "PRE"` spezifizieren
    - ⇨ `assert !s.isEmpty() : "PRE" ; // Specifying pre-condition`
  - Nachbedingung durch “normale” Java Assertion
    - ⇨ `assert s.isEmpty() || x.compareTo(s.getLast()) >= 0 ; // Post-condition`

- Beispiel
  - Klasse „SortedList“ (Code siehe nächste Folie)

T2-Beispiel: Spezifikation im Code

```
public class SortedList {

   private LinkedList<Comparable> s ;
   private Comparable max ;

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

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

   public void insert(Comparable x) {
      int i = 0 ;
      for (Comparable y : s) {
         if (y.compareTo(x) > 0) break ;
         i++ ; }
     s.add(i,x) ;
     if (max==null || x.compareTo(max) < 0) max = x ;
  }

  public Comparable get() {
     assert !s.isEmpty() : "PRE" ;                                                 // Pre-condition
     Comparable x = max ;
     s.remove(max) ;
     max = s.getLast() ;
     assert s.isEmpty() || x.compareTo(s.getLast()) >= 0 ;                         // Post-condition
     return x ;
  }
}
```

Testen mit T2

- Normalen JUnit-Test schreiben der aber lediglichT2 aufruft

```
import org.junit.Test;

public class MyTest {

    @Test
    public void test1() {

        // Call T2, pass the full name of the target class to it:
        Sequenic.T2.Main.Junit(SortedList.class.getName()) ;
    }
}
```

- Autor von T2
  - Wishnu Prasetya
- Weitere Informationen
  - http://code.google.com/p/t2framework/wiki/AutomatedTestingWithJunit

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) $S_t = \{\ldots, (w_i, p_{ij}, v_{ij}), \ldots\}$
  - Werte aller EIgenschaften aller Widgets zum Zeitpunkt t

- GUI Ereignis (Event) E
  - Initiatiiert transition aus Zustand S zu Zustand S’.

GUI-based Testing is Harder

Non-GUI tests

- Invoke methods
  - on test objects
  - created by test
- Provide input
  - arguments or global state
- Check expectations
  - return values
- Technology independent
  - Method call

GUI-based tests

- Trigger GUI events (e.g. clicks)
  - on existing GUI components
  - need to identify components!
- Provide input
  - filling text fields, …
- Check expectations
  - GUI structure / look
  - GUI behaviour
- Technology-Dependent
  - Java AWT, Eclipse SWT, Web, Android, Win, MacOs, …

GUI Testing Challenges

- Recording
  - Observing GUI states
  - Tracing GUI transitions
- Automated testing
  - State explosion problem
  - Explosion of event sequences that do the same thing
- Technology-dependency

➢ GUI test automation is hard

➢ GUI test maintenance is very hard

Capture and Replay

A capture and replay testing tool captures user sessions (user inputs and events) and stores them in scripts (one per session) suitable to be used to replay the user session.

An ad-hoc infrastructure is needed to intercept GUI events, GUI states, thus storing user sessions and also to be able to replay them.

- they can work at application or VM level

Recorded information

Inputs, outputs, and other information needed to replay a user session need to be recorded during the capture process.

- Examples:
  - General information: date/time of recording, etc.
  - System start-up information
  - Events from test tool to system
    - Point of control, event
  - Events from system to test tool
    - Checkpoints / expected outputs
  - Time stamps

Testen von GUIs: APIs

Klasse java.awt.Robot (ab JDK 1.3) erzeugt native Events zur Automatisierung von GUI-Aktionen

- Methoden: Tastatur-Events
  - keyPress(int keycode)
  - keyRelease(int keycode)
- Methoden: Maus-Events
  - mouseMove(int x, int y)
  - mousePress(long buttons)
  - mouseRelease(long buttons)
- Methoden: Timing
  - delay(int miliseconds)
  - setAutoDelay(int miliseconds)
- Methoden: Warten bis alle Events abgearbeitet
  - waitForIdle()
  - setAutoWaitforIdle(Boolean isOn)

Testen von GUIs: Freie Tools

WindowsTester

- Von Google gekauftes kommerzielles Werkzeug, das Eclipse geschenkt wurde (nun frei verfügbar)
- http://code.google.com/intl/de-DE/javadevtools/index.html

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

Capture and Replay: Process

1. The tester interacts with the system GUI to run the system, generating sequences of mouse clicks, UI and keyboard events
2. The tool captures and stores the user events and the GUI screenshots
   - ⇨ a script is produced per each user session
3. The tester can automatically replay the execution by running the script
   - ⇨ the script can be also changed by the tester
   - ⇨ the script can be enriched with expected output, checkpoints
   - ⇨ the script can be replicated to generate many variants (e.g., changing the input values)
4. In case of GUI changes, the script must be updated

Zusammenfassung

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