# Programmiertechnik II

## Modultests

# Ziele

- Überprüfung der Korrektheit eines Moduls
  - Korrektheit: Übereinstimmung mit (informaler) Spezifikation
  - Modul: kleine testbare Einheit (Funktion, Klasse)
  - Engl.: unit test
- White box testing
  - Annahme: Quelltext des zu testenden Codes ist verfügbar
  - Testfälle werden mit dem Ziel der Codeüberdeckung formuliert
  - Gegenteil: Black box testing (z.B. beta testing)
- Automatische Tests
  - Automatische Abwicklung aller Testfälle

# Probleme

- Test nur eines Moduls
  - Fehler, die sich aus der Kombination ergeben, sind nicht leicht erkennbar
  - Integrationstest, Systemtest
- Automatisierung ist schwierig
  - Entwicklung eines “Orakels” zur Bestimmung des Testergebnisses (verdict)
  - Aufbau einer Testumgebung als Voraussetzung für den Testlauf
- Auswahl der Testfälle
  - Codeüberdeckung (Zweigüberdeckung, Pfadüberdeckung, …)
  - Iterativ: Hinzufügen eines neuen Testfalls, wenn Fehler gefunden wurde
  - Ableitung aus Spezifikation: Ein Testfall pro geforderter Eigenschaft
  - Bildung von Äquivalenzklassen, testen eines Repräsentanten
- Abhängigkeit von Implementierung
  - Gray-box testing: Beschränkung auf “öffentliche” Operationen

# Begriffe

- Testfall (test case): ein elementarer Softwaretest
  - Vorbedingungen/Umgebung (fixture)
  - Eingaben
  - Erwartete Ausgaben
  - Erwartete Nachbedingungen
- Testziel (test purpose): Definition der zu testenden Eigenschaft
- Testergebnis (verdict): Wurde die Eigenschaft erfolgreich getestet?
  - Bestanden (pass), nicht bestanden (fail)
  - Unbestimmt (inconclusive)
  - Unerwartet bestanden/erwartet nicht bestanden (xpass, xfail)
- Testsuite: Sammlung von Testfällen

# Testen mit JUnit

- Java-Klassenbibliothek zum Testen (junit.sf.net)
  - junit.framework
- Kommandozeilentools zum Ausführen einer Testsuite
  - junit.textui, junit.swingui, junit.awtui: TestRunner
  - junit(1)
- Definition eines Testfalls
  - Ableiten von junit.framework.TestCase
  - Implementieren von Test-Methoden
  - Optional: Implementieren von runTest
  - Optional: Implementieren von setUp(), tearDown()
    - Test fixture
- Durchführen des Tests
  - Instanzieren der Testfallklasse
  - Aufruf von .run()
  - Alternativ: Integration mehrerer Testfälle in eine TestSuite

# Integration mehrerer Testfälle

- Methoden, deren Namen mit test beginnen, stellen Testfälle dar
- TestSuite.addTestSuite(java.lang.Class testClass)
  - Führt Introspection auf testClass aus

```
TestSuite suite = new TestSuite();
suite.addTestSuite(testClass.class);
```

- Testsuites können hierarchisch aufgebaut werden
- junit(1) erwartet statische Methode .suite()

# Bestimmung des Testergebnisses

- Aufruf von zu testenden Methoden mit Eingabeparameter
- Methoden assert* zur Festlegung des Ergebnisses
  - assertEquals(erwartet, beobachtet)
  - assertTrue(bedingung), assertFalse(bedingung)
  - assertNull(object), assertNotNull(object)
  - assertSame(erwartet, beobachtet), assertNotSame(e, b)
  - fail(fehlermeldung)
- Scheitern von assertion bedeutet Scheitern des Tests
- Unerwartete Ausnahmen sind Fehler im Test
  - Ausnahme abfangen, Behandlung durch fail()

# Parametrisierung von Tests

- Testfälle benötigen parameterlosen Konstruktor, oder müssen explizit instanziert werden
  - Explizite Konstruktion: ein Exemplar pro Testfall
  - Test fixture muss Testfall in definierten Ausgangszustand bringen
- Bei Suite-Konstruktion über Introspection keine Parametrisierung möglich
- Parametrisierung durch Ableitung:
  - Konstruktor oder setUp() der abgeleiteten Klasse führen Parametrisierung durch

# JUnit 4

- “Alte” APIs stehen weiter zur Verfügung
- Deklaration von Testfällen über Java-5-Annotationen möglich
  - @Test public void good_primes()...
  - Parameter expected= für erwartete Ausnahmen
- Damit: Ableitung von TestCase nicht mehr nötig
  - aber: Woher kommt assertTrue/assertEquals/...
  - Variante 1: org.junit.Assert.assertEquals(...)
  - Variante 2: import static org.junit.Assert.assertEquals;
- Weitere Annotationsklassen für fixture, ...
  - @Before, @After, @Ignore, ...