# Unit Testing mit JUnit

Dr. Andreas Schroeder

# Überblick

Was dieses Video behandelt

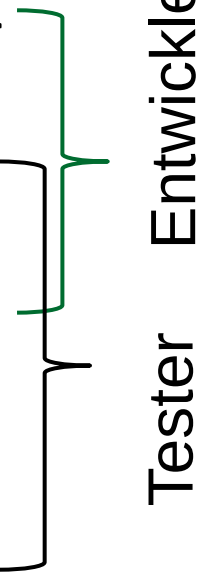
- Warum Testen?
- Was sind Unit Tests?
- Der Teufelskreis des Nicht-Testens
- JUnit
- Unit Test Vorteile
- Test-Inspiration
- Wann aufhören?

# Warum Testen?

- Wie überprüft man, dass ein System das Richtige tut?
  - System in richtigen Zustand bringen
  - Mit Beispiel-Eingaben ausführen
  - Überprüfen ob das erwartete Verhalten auftritt
- Wie oft muss man überprüfen?
  - Direkt beim Einführen einer neuen Funktionalität
  - Jede Änderung am System kann anderes gewünschtes Verhalten beeinflussen
  - Nach jeder Änderung muss das System überprüft werden!
- … kann man nicht ein Programm dafür schreiben?
  - Man kann! Und zwar **automatisierte** Tests

# Was sind Unit Tests? (1/2)

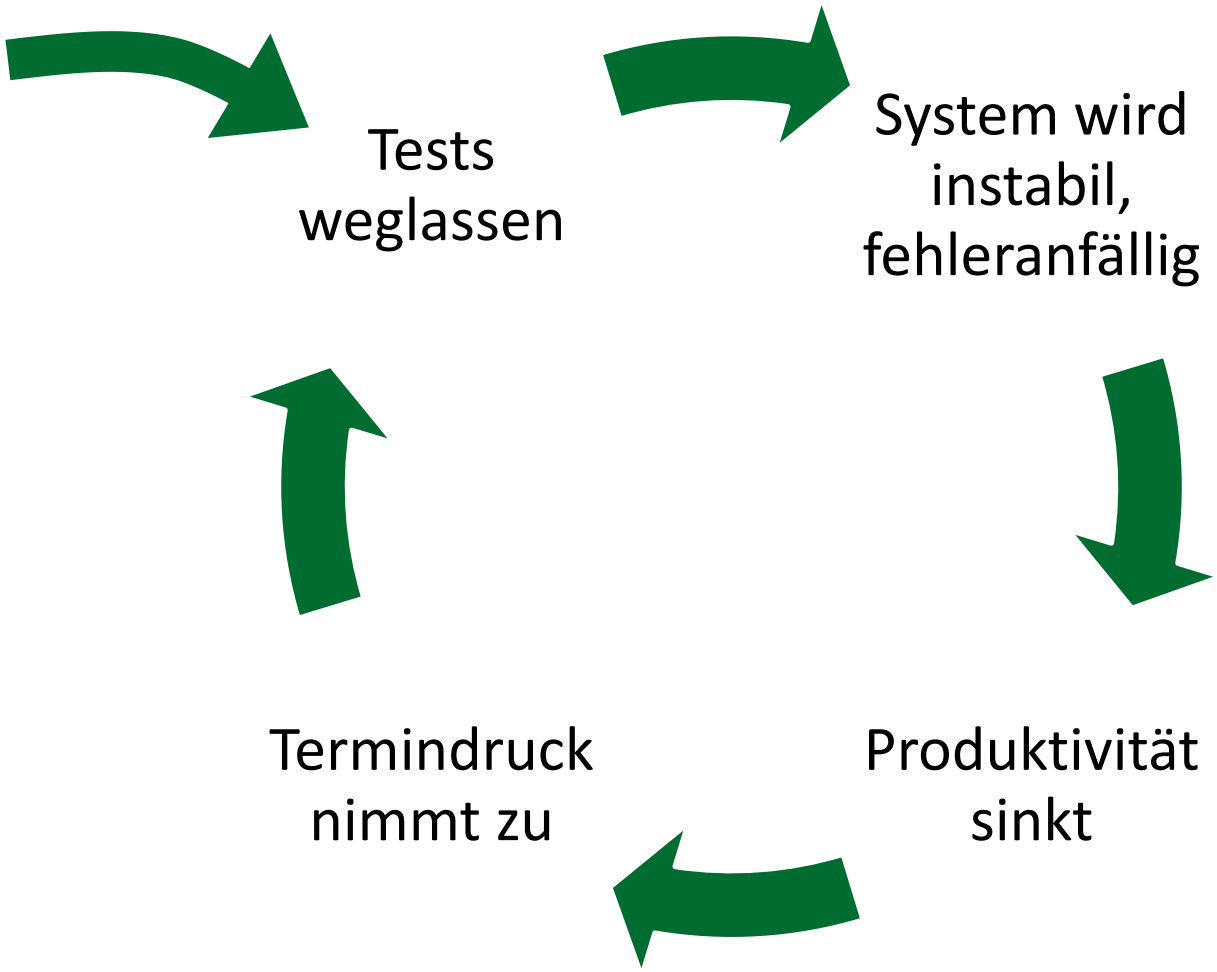
- Test-Programme sind **automatisierte Tests**
  - Genauer: die einzelnen Testfälle werden Test genannt
- Tests können unterschiedlich fein sein
  - **Unit Tests** testen die kleinste testbare Einheit einer Software (z.B. eine einzelne Methode in Java).
  - **Integrationstests** testen die Interaktion zwischen Komponenten (z.B. public Interfaces).
  - **Systemtests** testen die Software als Ganzes.
  - **Systemintegrationstests** testen ob die Software richtig in seiner Umgebung integriert ist.



# Was sind Unit Tests? (2/2)

- Jede Code-Änderung kann bestehende Funktionalität kaputt machen
- Angenommen, es gäbe für jedes gewünschte Verhalten einen Test, dann könnte man…
  - per Knopfdruck überprüfen ob eine Änderung eine bestehende Funktionalität kaputt gemacht hat
  - aus dem Test herauslesen, welches Verhalten das System nicht mehr zeigt (evtl. mit dem Debugger)
  - den Code reparieren, so dass der Test wieder erfolgreich durchläuft

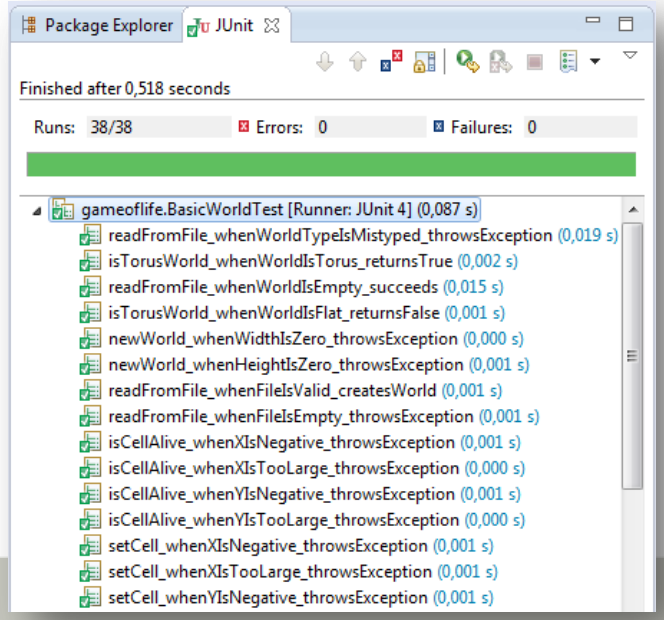
# Der Teufelskreis des Nicht-Testens



# JUnit

JUnit.org

- JUnit ist ein Unit Testing Framework für Java
- JUnit ist in Eclipse integriert
  - Runner und View geben Überblick über Testergebnisse
- Ein Test kann zwei Ergebnisse produzieren:
  - **Bestanden (Grün)** oder **Fehlgeschlagen (Rot)**
- Fünf Schritte zum Unit-Test
  1. Fixture erzeugen
  2. Eingabe erzeugen
  3. Test Ausführen
  4. Verhalten und Zustand überprüfen
  5. Aufräumen



# Wiederholung des gezeigten

- Test-Methode mit @Test annotieren
- Gemeinsame Fixture in @Before-Methode auslagern
- Sprechende Benennung der Test-Methoden:
  Methode_HatEinenEffekt_UnterBestimmtenBedingungen
- Bestandteile eines Tests klar trennen
  - Vorbereiten
  - Durchführen
  - Prüfen
  - Aufräumen (in @After-Methode)
- Nur ein Verhalten in einer Methode testen
  Unterschiedliche Ausführungspfade in unterschiedliche Tests!

# JUnit-Annotationen

| Annotation | Bedeutung |
|---|---|
| @Test | Test-Methode |
| @Before | Methode vor jedem Test ausführen |
| @After | Methode nach jedem Test ausführen |
| @BeforeClass | Statische Methode vor allen Tests ein mal ausführen |
| @AfterClass | statische Methode nach allen Tests ein mal ausführen |
| @Ignore | Ignoriere Test-Methode |

# Assertions

- Parameter-Schema von Assertions
  - Nachricht bei Misserfolg, erwarteter Wert, erhaltener Wert

| Assertion | Funktionsweise |
|---|---|
| fail | Schlägt fehl |
| assertTrue | Überprüft ob die Bedingung wahr ist |
| assertEquals | Überprüft ob die Objekte gleich sind (mit equals) |
| assertNotSame | Überprüft dass die Referenzen auf unterschiedliche Objekte zeigen (unabhängig von equals) |
| assertNull | Überprüft dass Parameter null ist |
| assertNotNull | Überprüft dass Parameter nicht null ist |
| assertArrayEquals | Überprüft ob zwei Arrays gleich sind |

# Unit Test Vorteile

- Unit Tests ändern den **Blickwinkel**
  - Selbst geschriebener Code wird selbst verwendet, Probleme an der externen Schnittstelle werden deutlich
- Unit Tests schaffen **Vertrauen**
  - Sie zeigen dass die Code-Basis die gewünschte Funktionalität tatsächlich umsetzt
- Unit Tests sind ausführbare **Dokumentation** und **Spezifikation**
  - Sie zeigen was vom System erwartet wird, und dokumentieren die Schnittstellen

# AssertThat-Variante

- assertThat Paramter
  - Beschreibung (optional)
  - Tatsächlicher Wert
  - Bedingungen für den Wert
- Beispiel 1: theBiscuit gleich myBiscuit

  assertEquals(myBiscuit, theBiscuit)

  assertThat(theBiscuit, is(equalTo(myBiscuit)))
- Beispiel 2:  list.size() ist 5

  assertEquals(5, list.size())

  assertThat(list.size(), is(5))

# assertThat Matcher

| Matcher | Funktionsweise |
|---|---|
| anything | Matcht alles |
| is | Verbessert nur Lesbarkeit, z.B. is(equalTo(item)) |
| equalTo | Überprüft Gleichheit mit equals-Methode |
| sameInstance | Überprüft Objekt-Identität |
| allof, anyof, not | Logische Junktoren für Matchers (und, oder, nicht) |
| instanceOf | Überprüft Typ des Objekts |
| notNullValue, nullValue | Überprüft dass Wert (nicht) Null ist. |
| hasItem, hasItems | Überprüft ob ein Array die genannten Elemente enthält |

https://code.google.com/p/hamcrest/wiki/Tutorial

# Test-Inspiration

Ein paar Ideen für tests:

- **Haupt-Funktionalität** (d.h. teste dass der haupt-pfad geht)
- **Zweig-basiertes Testen** (d.h. prüfe dass es einen Test für jeden Zweig jeder Bedingung gibt)
- **Korrekte Fehlerbehandlung** (d.h. prüfe dass Methoden null-Parameter, geschlossene Ressourcen, und fehlgeschlagene Verbindungen behandeln)
- **Funktioniert wie beschrieben** (d.h. falls die Dokumentation Regeln für einen Methodenaufruf beschreibt, teste diese Regeln)

# Wann aufhören?

- Tests dienen dazu, **Vertrauen** in die Code-Basis zu schaffen
- Wenn das Team Vertrauen in den eigenen Code hat, dann wurde genug getestet
- Jeder im Team muss auf sein eigenes Vertrauensgefühl hören – bezüglich des Codes des gesamten Teams

# Noch ein Gedanke…

Fear leads to anger, anger leads to hate, hate leads to suffering.

**No tests lead to fear.**

*Dr. Philip Mayer*

# Zusammenfassung

Was in diesem Video behandelt wurde

- Warum Testen?
- Was sind Unit Tests?
- Der Teufelskreis des Nicht-Testens
- JUnit
- Unit Test Vorteile
- Test-Inspiration
- Wann aufhören?