Testen und Validieren mit JUnit und NUnit

Nicolas Schmidt

Fachbereich Informatik
Universität Ulm

22. Mai 2025

Inhaltsverzeichnis

Motivation

- Komplexität moderner Softwaresysteme nimmt ständig zu.
- Fehler im Produkt können hohen wirtschaftlichen und sicherheitsrelevanten Schaden verursachen.
- Automatisiertes Testen erhöht Entwicklungsqualität und -effizienz.
- Vorstellung von zwei etablierten Frameworks: JUnit (Java) und NUnit (.NET).

Ziele und Aufbau

- **Ziel:** Verständnis der Konzepte des Softwaretestens, praktische Anwendung mit JUnit und NUnit sowie Best Practices.
- Aufbau:
  1. Theoretische Grundlagen des Softwaretestens
  2. Praktische Umsetzung mit JUnit und NUnit
  3. Analyse von Testergebnissen und Testmetriken
  4. Weiterführende Themen: TDD, Mocking, CI/CD
  5. Zusammenfassung und Ausblick

Grundlagen

Definition

Softwaretest ist ein Prozess zur Bewertung der Qualität eines Softwareprodukts durch das Ausführen von Testfällen, um Fehler aufzudecken und die Anforderungen zu validieren.

Grundlagen

Definition

Softwaretest ist ein Prozess zur Bewertung der Qualität eines Softwareprodukts durch das Ausführen von Testfällen, um Fehler aufzudecken und die Anforderungen zu validieren.

- **Testarten:**
  - *Unit-Tests:* kleinste Bausteine (Methoden/Klassen) isoliert testen
  - *Integrationstests:* Zusammenspiel mehrerer Module prüfen
  - *Systemtests:* Gesamtsystem im realen Einsatzszenario testen
  - *Abnahmetests:* Tests aus Sicht des Kunden
- **Testprinzipien:**
  - Black-Box-Tests (funktional, ohne Kenntnis der Implementierung)
  - White-Box-Tests (strukturorientiert, mit Kenntnis des Codes)
- **Verifikation vs. Validierung:**
  - Verifikation: „Bauen wir das Produkt richtig?“
  - Validierung: „Bauen wir das richtige Produkt?“

Testprozess und Testpyramide

Testprozess:

1. Testplanung
2. Testentwurf
3. Testimplementierung
4. Testdurchführung
5. Auswertung

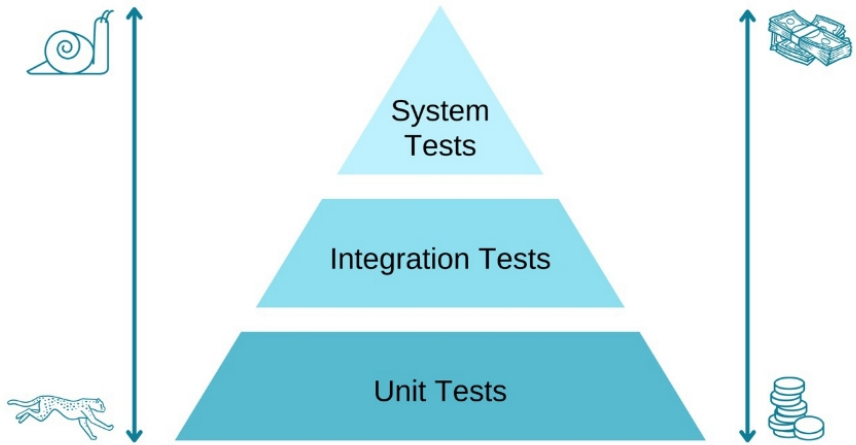
Testprozess und Testpyramide

Testprozess:

1. Testplanung
2. Testentwurf
3. Testimplementierung
4. Testdurchführung
5. Auswertung

Testpyramide (Mike Cohn):

- Viele Unit-Tests (Basis)
- Weniger Integrationstests (Mitte)
- Wenige GUI- und End-to-End-Tests (Spitze)



Qualitätsmerkmale von Tests

- **Automatisierbarkeit:** Tests müssen automatisch ausführbar sein.
- **Reproduzierbarkeit:** Tests sollten immer das gleiche Ergebnis liefern.
- **Unabhängigkeit:** Tests sollen unabhängig voneinander laufen.
- **Wartbarkeit:** Tests müssen einfach zu verstehen und anzupassen sein.
- **Schnelligkeit:** Tests sollen möglichst schnell durchlaufen.

JUnit – Überblick

- JUnit ist das populärste Unit-Test-Framework für Java.
- Aktuelle Version: JUnit 5 (Jupiter API).
- Ermöglicht einfache Definition von Testfällen mit Annotationen.
- Integration in IDEs (IntelliJ, Eclipse) und Build-Tools (Maven, Gradle).
- Unterstützt Assertions, Setup/Teardown, Parameterized Tests und mehr.

JUnit Beispiel

```
import org.junit.jupiter.api.Test;
import static org.junit.jupiter.api.Assertions.*;

class CalculatorTest {
  @Test
  void add() {
    Calculator calc = new Calculator();
    assertEquals(5, calc.add(2, 3), "Addition sollte 5
      ergeben");
  }
}
```

Beispiel mit einfachem Additionstest.

JUnit Beispiel (Fortsetzung)

```
@Test
void divideByZero() {
  Calculator calc = new Calculator();
  assertThrows(ArithmeticException.class, () -> calc.divide
    (10, 0));
}
```

Beispiel mit Exception-Test.

NUnit – Überblick

- NUnit ist ein populäres Unit-Test-Framework für .NET Sprachen.
- Unterstützt Attribute wie **[Test]**, **[TestFixture]**, **[SetUp]**.
- Integration in Visual Studio und Build-Systeme (MSBuild, Azure DevOps).
- Umfangreiche Assertion-Library, Testparameter, Setup/Teardown.
- Unterstützt Testkategorien und parallelisierte Testausführung.

NUnit Beispiel

```
using NUnit.Framework;
using System;

[TestFixture]
public class CalculatorTest {
  [Test]
  public void Add() {
    var calc = new Calculator();
    Assert.AreEqual(5, calc.Add(2, 3));
  }

  [Test]
  public void DivideByZero_ShouldThrow() {
    var calc = new Calculator();
    Assert.Throws<DivideByZeroException>(() => calc.Divide
      (10, 0));
  }
}
```

Einfaches Beispiel mit normalem Test und Exception-Test.

Best Practices für Unit-Tests

- Schreibe unabhängige, deterministische Tests.
- Teste nur eine Funktionalität pro Testfall.
- Nutze aussagekräftige Testnamen.
- Verwende Setup- und Teardown-Methoden für wiederkehrende Vorbedingungen.
- Nutze Mocks und Stubs, um Abhängigkeiten zu isolieren.
- Teste auch Fehler- und Ausnahmefälle.

Mocking – Konzept und Nutzen

- Abhängigkeiten (z.B. Datenbanken, Webservices) werden durch Mock-Objekte ersetzt.
- Erlaubt isoliertes Testen der Unit ohne externe Effekte.
- Beliebte Frameworks: Mockito (Java), Moq (.NET).
- Beispiel: Mocking eines Datenbankzugriffs in JUnit.

Beispiel Mocking mit Mockito (JUnit)

```
import static org.mockito.Mockito.*;
import org.junit.jupiter.api.Test;

class UserServiceTest {
  @Test
  void testGetUserName() {
    UserRepository repo = mock(UserRepository.class);
    when(repo.findUser(1)).thenReturn(new User(1, "Nicolas"
      ));

    UserService service = new UserService(repo);
    String name = service.getUserName(1);

    assertEquals("Nicolas", name);
  }
}
```

Beispiel Mocking mit Moq (NUnit)

```
using Moq;
using NUnit.Framework;

[TestFixture]
public class UserServiceTest {
  [Test]
  public void TestGetUserName() {
    var repoMock = new Mock<IUserRepository>();
    repoMock.Setup(r => r.FindUser(1)).Returns(new User(1,
      "Nicolas"));
    var service = new UserService(repoMock.Object);
    var name = service.GetUserName(1);

    Assert.AreEqual("Nicolas", name);
  }
}
```

Testmetriken und Qualitätssicherung

- **Testabdeckung (Coverage):** Wie viel Prozent des Codes werden von Tests geprüft? (Zeilen, Branches)
- **Testlaufzeiten:** Tests sollten möglichst schnell sein, um Feedbackzyklen kurz zu halten.
- **Fehlerrate:** Anzahl der fehlschlagenden Tests vs. Gesamtanzahl.
- **Mutation Testing:** Techniken zur Messung der Testqualität durch absichtliche Codeänderungen.

Testberichte und Automatisierung

- JUnit erzeugt XML-Reports, die in CI/CD-Systeme (Jenkins, GitHub Actions) eingebunden werden können.
- NUnit bietet ebenfalls XML-Reports und Konsolenausgabe.
- Automatisierte Tests in CI/CD erhöhen die Zuverlässigkeit und Geschwindigkeit der Softwareentwicklung.
- Beispiel CI-Pipeline mit automatisiertem Testlauf.

Testgetriebene Entwicklung (TDD)

- Zyklen: *Red – Green – Refactor*
- Schreibe zuerst einen fehlgeschlagenen Test, implementiere dann Code, bis Test besteht.
- Vorteile: Bessere Testabdeckung, saubere Schnittstellen, weniger Bugs.
- JUnit und NUnit eignen sich hervorragend für TDD.

Continuous Integration und Testautomatisierung

- Automatischer Testlauf bei jedem Commit.
- Sofortiges Feedback bei fehlschlagenden Tests.
- Tools: Jenkins, Travis CI, Azure DevOps, GitHub Actions.
- Ermöglicht schnelle und sichere Releases.

Herausforderungen und Grenzen

- Tests sind nur so gut wie ihre Testfälle.
- Übermäßige Tests erhöhen Wartungsaufwand.
- Schwierigkeit bei UI-Tests und nicht-deterministischen Systemen.
- Flaky Tests: intermittierende Fehler erschweren Automatisierung.
- Wichtig: Teststrategie immer an Projekt und Team anpassen.

Zusammenfassung

- JUnit und NUnit sind leistungsfähige Frameworks für Unit-Tests in Java bzw. .NET.
- Softwaretests sind zentral für Qualitätssicherung und müssen systematisch durchgeführt werden.
- Mocking, TDD und CI/CD sind wichtige ergänzende Techniken.
- Tests sollten wartbar, automatisierbar und aussagekräftig sein.

Ausblick

- Weitere Testansätze: Property-Based Testing (z.B. jqwik, FsCheck)
- Mutation Testing zur Testqualitätsbewertung
- Testen von verteilten Systemen und Microservices
- KI-gestütztes Testen
- Qualitätssicherung im agilen und DevOps-Umfeld

Vielen Dank für Ihre Aufmerksamkeit!

Gibt es Fragen?