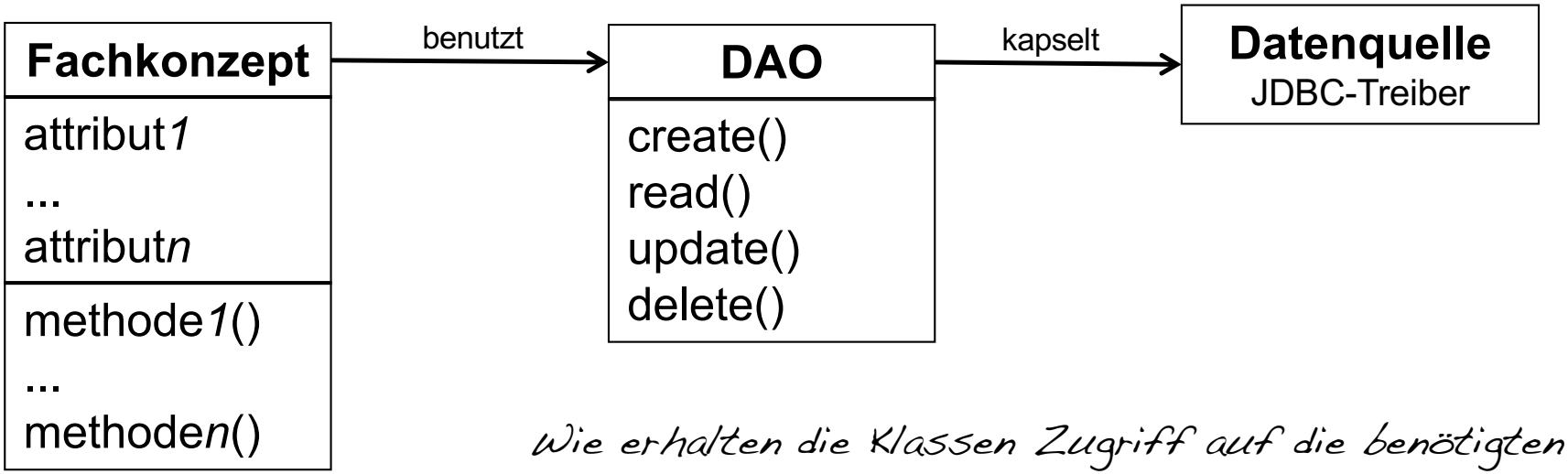
- Testbarkeit
  - Häufig ist es selbst bei einem Unit-Test nicht möglich, eine zu testende Klasse völlig isoliert zu betrachten
  - Beispiel: Es soll eine Fachkonzeptklasse getestet werden
    a. die verwendete DAO-Klasse ist noch nicht fertiggestellt
    b. das DAO soll nicht die produktive Datenbank verwenden, sondern die Datenbank in der Entwicklungsumgebung



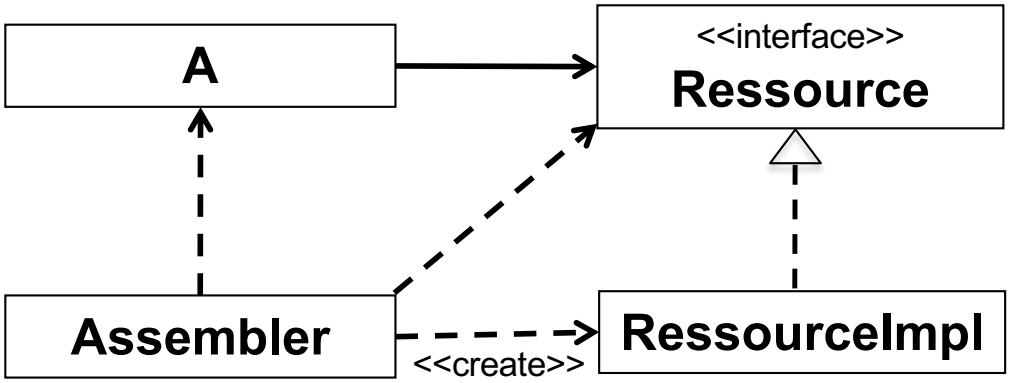
Wie erhalten die Klassen Zugriff auf die benötigten Ressourcen?

- Es gibt im Wesentlichen drei Möglichkeiten, wie ein Objekt *A* eine Referenz auf ein benötigtes Objekt *B* erhält
  1. A erzeugt selber die Instanz B
  2. Dependency Injection
  3. Verwendung eines Namensdienstes (Service Locator)

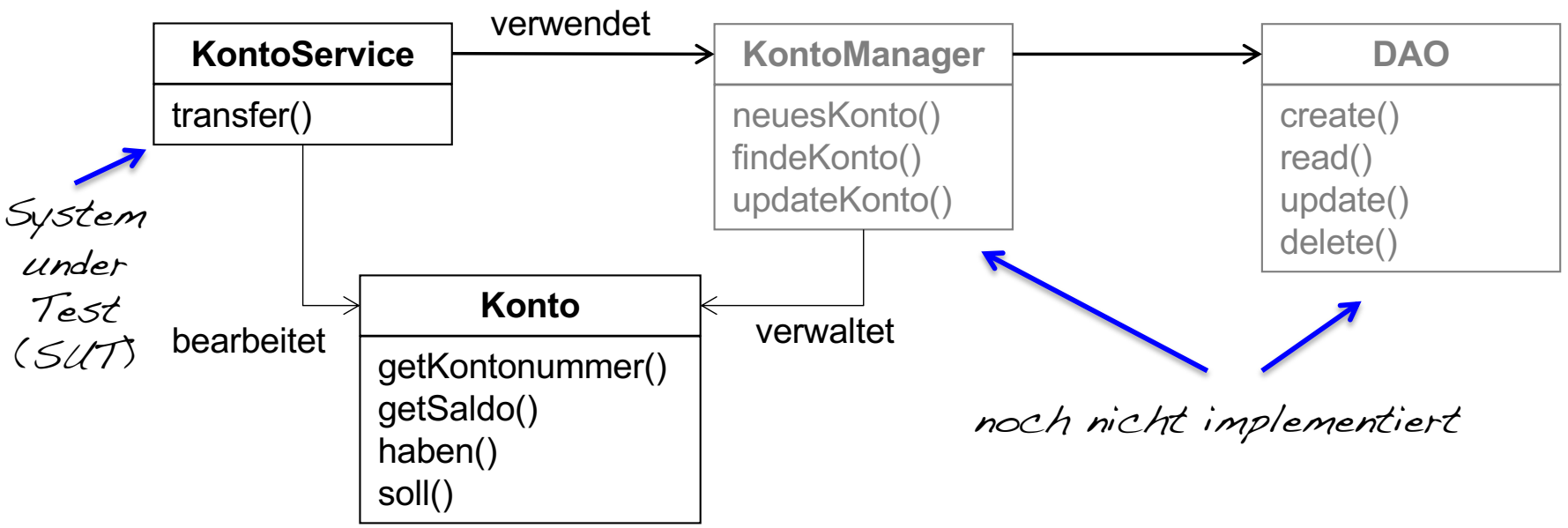
Einfluss auf die Testbarkeit?

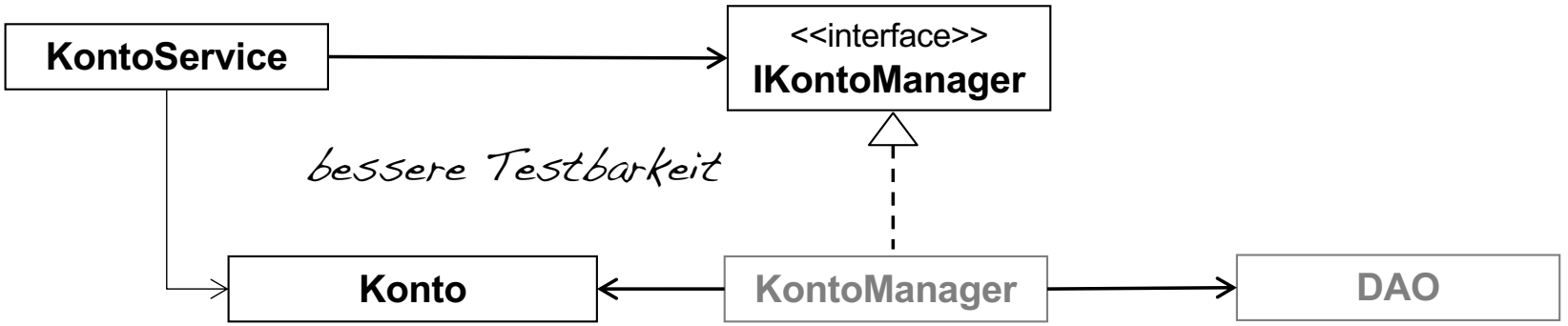
- Die Testbarkeit wird deutlich verbessert, wenn in einer zu testenden Klasse Assoziationen zu anderen Objekten nicht aktiv aufgebaut werden

- Dependency Injection (DI)
  - Eine höhere Flexibilität bezüglich der Umwelt wird erreicht, wenn Assoziationen zu anderen Objekten von außen injiziert werden
  - Ein Objekt kümmert sich also nicht mehr aktiv um eine Ressource, sondern bekommt die entsprechende Referenz von außen übergeben (von einem "Assembler")

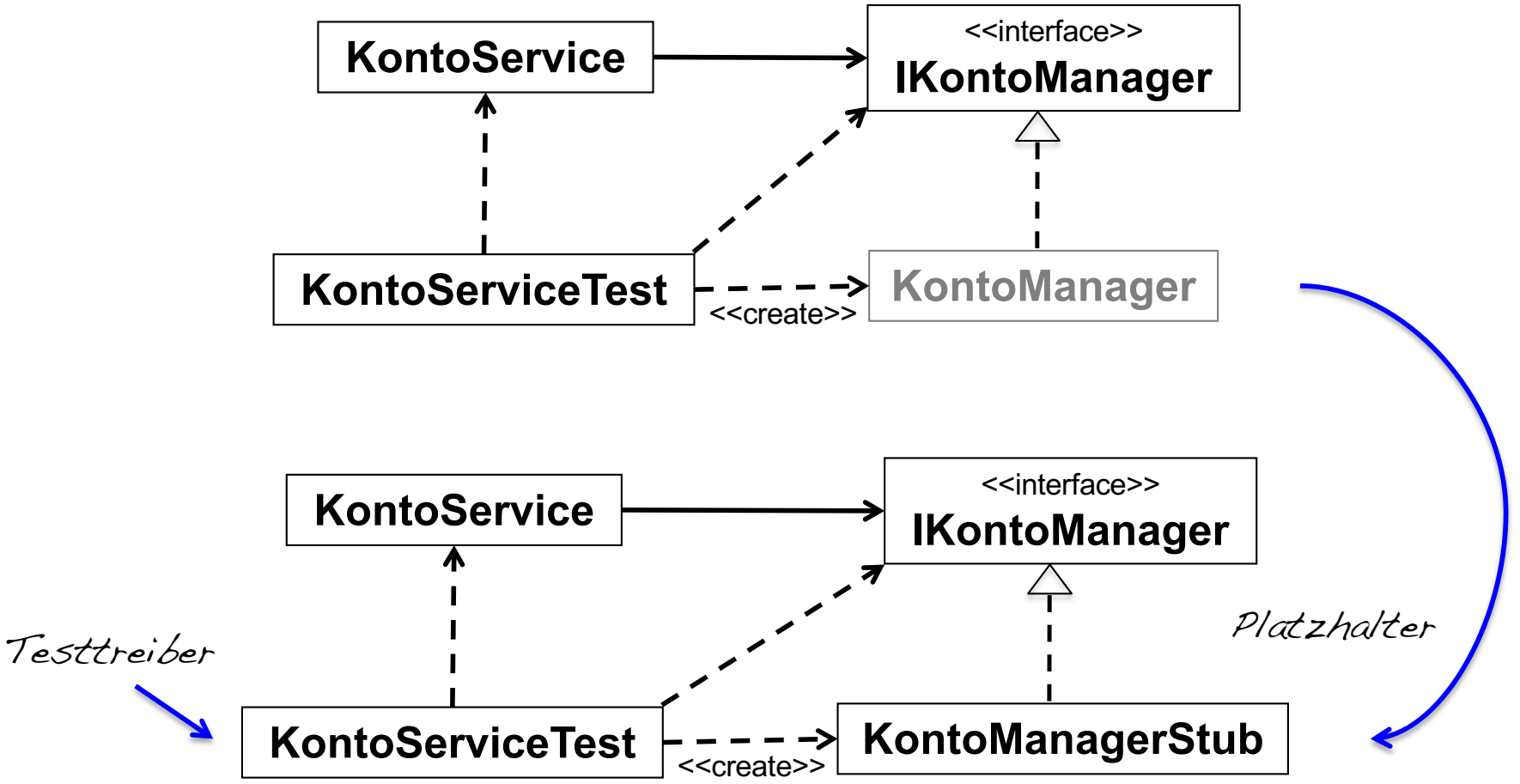


Fallbeispiel: Kontenverwaltung





– Einsatz von DI (am Beispiel eines JUnit-Tests)



- Die „Nebenklasse“ Konto (indirekter Test)

```
public class Konto {

    private String kontoNummer;
    private double saldo;

    public Konto(String kontoID, double saldo){
        this.kontoNummer = kontoID;
        this.saldo = saldo;
    }
    public void soll(double umsatz){
        saldo -= umsatz;
    }
    public void haben(double umsatz){
        saldo += umsatz;
    }
    public String getKontoNummer() {
        return kontoNummer;
    }
    public double getSaldo() {
        return saldo;
    }
}
```

- Die Schnittstelle `IKontoManager`

```
public interface IKontoManager {
    Konto findeKonto(String kontoNummer);
    void updateKonto(Konto konto);
    void neuesKonto(Konto konto);
}
```

- Implementierung der Klasse `KontoService` ist zu testen

```
public class KontoService {

   private IKontoManager kontoManager;

   public void setKontoManager(IKontoManager kontoManager){
      this.kontoManager = kontoManager;
   }

   public void transfer(String quelle, String ziel, double umsatz){
      Konto quelleKonto = kontoManager.findeKonto(quelle);
      Konto zielKonto = kontoManager.findeKonto(ziel);

      quelleKonto.soll(umsatz);
      zielKonto.haben(umsatz);

      kontoManager.updateKonto(quelleKonto);
      kontoManager.updateKonto(zielKonto);
    }
}
```

Programmierung gegen Interface

Instanz von KontoManager wird nicht selber erzeugt

- Platzhalter (Stub) für die Klasse `KontoManager`

```
public class KontoManagerStub implements IKontoManager{

   private Konto konto1;
   private Konto konto2;

   public void neuesKonto(Konto konto){
      if (konto1==null) konto1 = konto;
      konto2 = konto;
   }

   public void updateKonto(Konto konto){
   }

   public Konto findeKonto(String kontoNummer){
      if (konto1.getKontoNummer().equals(kontoNummer)) return konto1;
      if (konto2.getKontoNummer().equals(kontoNummer)) return konto2;
      throw new IllegalArgumentException();
   }
}
```

Rudimentäre Implementierung:

Es können nur zwei Konten „verwaltet" werden

- Testtreiber für die Klasse `KontoService`

```
public class KontoServiceTest {

   @Test
   public void testTransfer() {

      Konto quelle = new Konto("1", 1000.0);
      Konto ziel = new Konto("2", 100.0);
      IKontoManager stub = new KontoManagerStub();
      stub.neuesKonto(quelle);
      stub.neuesKonto(ziel);

      KontoService service = new KontoService();
      service.setKontoManager(stub);

      service.transfer("1", "2", 500.0);

      assertEquals(500.0, quelle.getSaldo(),0.0);
      assertEquals(600.0, ziel.getSaldo(), 0.0);
   }
}
```

Erzeugung und Konfiguration des Platzhalters

DI des Platzhalters

Aufruf der zu testenden Methode

Zustand überprüfen

Stubs vs Mocks

- Testen mit *Stubs*
  - Mit *Stubs* werden die zu testenden Einheiten von der Umgebung entkoppelt
  - *Stub* simuliert für den Testfall erforderliche Fachlogik
  - Zustandstest (Test der Zustandsänderung)

- Testen mit Mocks
  - Mocks sorgen ebenfalls für eine Entkopplung
  - Mocks kontrollieren die Anzahl der Methodenaufrufe
  - Mocks kontrollieren die Abfolge der Methodenaufrufe
  - Mocks enthalten keine/begrenzte Simulation der Fachlogik
  - Verhaltenstest

- Testen mit Mocks
  - Da Mocks keine Fachlogik enthalten, können sie aus einer Schnittstellendefinition automatisch generiert werden
  - Dies erfordert die Unterstützung durch entsprechende Werkzeuge /Frameworks (z.B. *EasyMock*)
  - Damit kann der Aufwand bei der Testimplementierung reduziert werden

- *EasyMock*-Framework
  - `easymock.jar` wird in den `classpath` der Anwendung aufgenommen
  - Aus einem Interface wird im Testtreiber mit der Methode `createMock` zur Testlaufzeit ein Mock-Objekt generiert
  - Das Mock-Objekt befindet sich dann automatisch in einer Aufzeichnungsphase

- Es werden alle, während des eigentlichen Tests erwarteten, Methodenaufrufe durchgespielt und vom Mock-Objekt aufgezeichnet
- Danach wird das Mock-Objekt in einen *Replay*-Modus versetzt
- Der eigentliche Testfall wird gestartet
- In einem *Verify*-Modus vergleicht das Mock-Objekt dann die aufgezeichneten Methodenaufrufe mit den tatsächlich im Test erfolgten Methodenaufrufen
- Abweichungen führen zum Scheitern des JUnit-Tests

```
public class KontoServiceEasyMockTest {
   private IKontoManager mock;
   @BeforeEach
   public void setUp(){
      mock = mock("mock", IKontoManager.class);
   }
   @Test
   public void test() {
      Konto quelle = new Konto("1", 1000.0);
      Konto ziel = new Konto("2", 100.0);
      expect(mock.findeKonto("1")).andReturn(quelle);
      expect(mock.findeKonto("2")).andReturn(ziel);
      mock.updateKonto(quelle);
      mock.updateKonto(ziel);
      replay(mock);
      KontoService service = new KontoService();
      service.setKontoManager(mock);
      service.transfer("1", "2", 500.0);
   }
   @AfterEach
   public void tearDown(){
      verify(mock);
   }
}
```

generiert Mock-Objekt aus Interface

Aufzeichnung des erwarteten Verhaltens

Aufruf der zu testenden Methode

Prüfen, ob erwartete Aufrufe nicht erfolgt sind

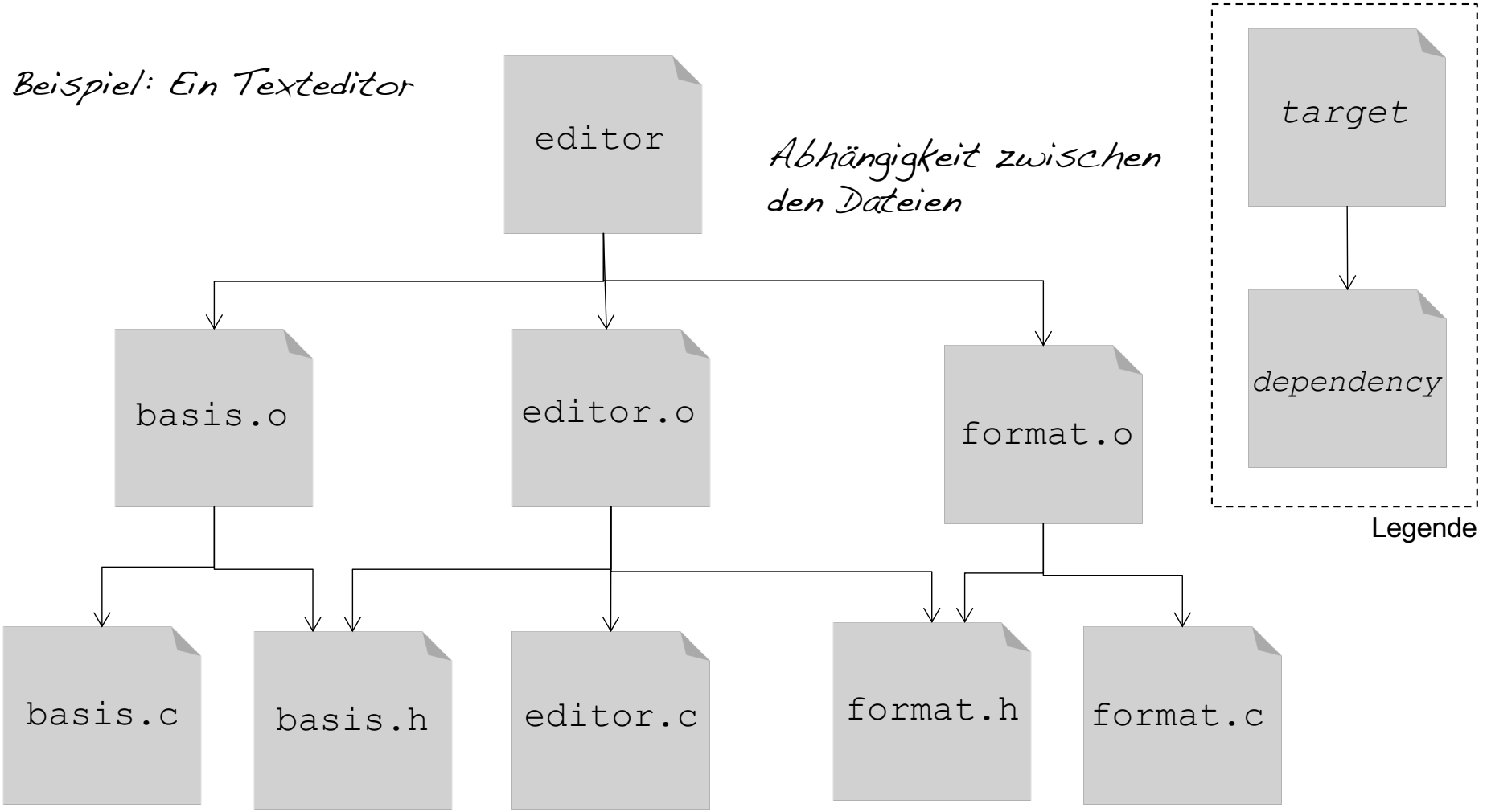
- Der Verhaltenstest mit einem Mock kann im obigen Fall noch um einen Zustandstest angereichert werden

```
//...
service.transfer("1", "2", 500.0);

assertEquals(500.0, quelle.getSaldo(),0.0);
assertEquals(600.0, ziel.getSaldo(), 0.0);
```

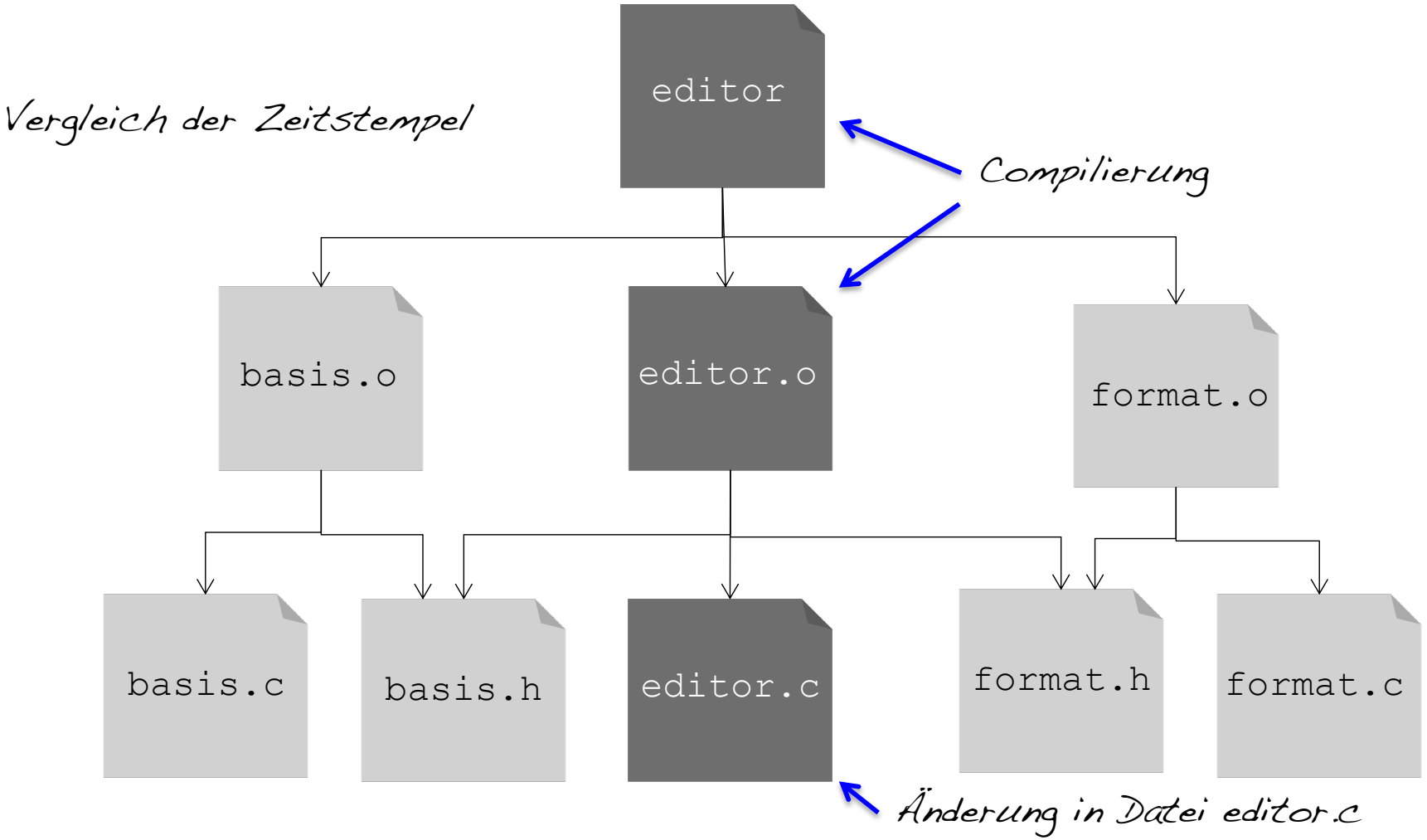
Build-Automatisierung

- Die (allgemeine) Automatisierung haben wir bereits als konstruktive Maßnahme der Qualitätssicherung identifiziert
- Wir betrachten hier als wichtigen Spezialfall die Automatisierung des Build-Prozesses
- Obwohl eine IDE in der Regel einen grundlegenden Build-Prozess bietet, sollte ein unabhängiger Build-Prozess aufgesetzt werden
- Dazu wird ein geeignetes Build-Tool benötigt:
  - make (C, C++, ...)
  - Ant (Java)
  - Maven (Java)
  - ...

– Neben der Automatisierung des Build-Prozesses bieten die Werkzeuge auch eine inkrementelle (bedarfsgerechte) Compilierung



– Bei einer nachträglichen Änderung in der Datei `editor.c` müssen die Dateien `basis.o` und `format.o` nicht neu übersetzt werden



- make
  - Bedingungsgesteuerte Ausführung von Shell-Skript-Kommandos
  - `make` steuert u.a. die folgenden Aktivitäten
    - Kopieren von Dateien
    - Inkrementelle Compilierung
    - Linken
  - Der Erstellungsprozess wird formal in einem *Makefile* beschrieben
  - Im *Makefile* sind alle Abhängigkeiten beschrieben und alle Aktivitäten definiert
  - Ein *Makefile* besteht aus Regeln der folgenden Form

```
Ziel : Voraussetzung ... ...
    Befehl
    ...
    ...
```

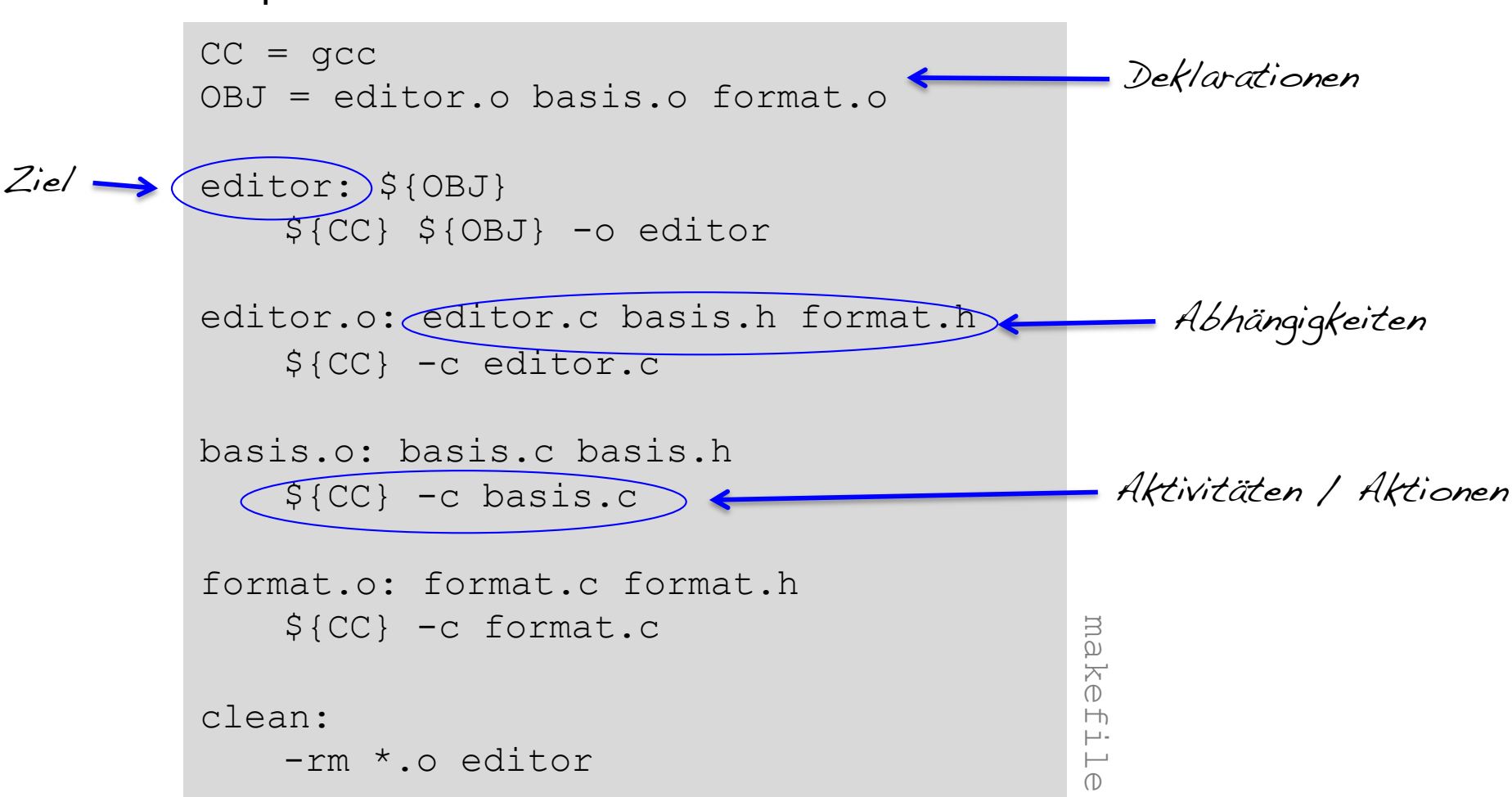
makefile

Jede Befehlszeile muss mit einem Tabulatorzeichen beginnen

- Beispiel: Editor

```
CC = gcc
OBJ = editor.o basis.o format.o

editor: ${OBJ}
    ${CC} ${OBJ} -o editor

editor.o: editor.c basis.h format.h
    ${CC} -c editor.c

basis.o: basis.c basis.h
    ${CC} -c basis.c

format.o: format.c format.h
    ${CC} -c format.c

clean:
    -rm *.o editor
```

makefile



- Mit `make` wird auf der Kommandozeilenebene die Verarbeitung der Datei `makefile` gestartet

- Ant
  - Da `make` die Ausführung von beliebigen Shell-Skript-Kommandos erlaubt, ist `make` plattformabhängig
  - Die Alternative *Ant* kommt im Java-Umfeld zum Einsatz und bietet eine angemessene Plattformunabhängigkeit
  - Die Konfigurationsdatei `build.xml` ähnelt dem `makefile`, wird aber im standardisierten XML-Format beschrieben
  - In der `build.xml` wird beschrieben, wie Ziele (*targets*), unter der Berücksichtigung von Abhängigkeiten (*dependencies*), durch die Ausführung von Kommandos erreicht werden können
  - Grundstruktur

```
<project name="..." default="..." basedir=".">
   <target name="..." depends="...">
      <command />
   </target>
</project>
```

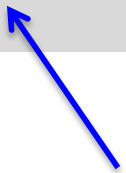
build.xml

- Attribute des Elements `project`:

Attribut	Bedeutung
`name`	Name des Projekts
`default`	Wird ausgeführt wenn Ant ohne Target aufgerufen wird
`basedir`	Basisverzeichnis für alle Targets

- Die Teilschritte eines Build-Prozesses werden als *Targets* bezeichnet
- Ein *Target* besitzt einen Namen und evtl. Abhängigkeiten zu anderen Targets

```
<target name="A" depends="B, C">
      <command />
</target>
```



Bevor Target A ausgeführt wird, werden zunächst die Targets B und C abgearbeitet

- Ein *Target* kann eine Folge von Kommandos (*Tasks*) enthalten
- *Ant* enthält über 100 *Tasks*
  - javac
  - mkdir
  - copy
  - junit
  - jar
  - cvs
  - ...

1. Einstellungen

- Definition von Variablen mit dem Target `property`
  - Einstellungen können zentral vorgenommen werden

```
<property name="key" value="wert" />
```

  - Die Zeichenfolge `${key}` wird dann in den `wert` aufgelöst

- Zugriffspfade auf externe Klassenbibliotheken setzen

```
<path id="project.classpath">
   <pathelement path="./${lib.dir}/log4j.jar" />
</path>
```

2. Vorbereiten der Verzeichnisse

- Verzeichnisse anlegen

```
<target name="prepare">
   <echo message="creating: ${build.dir}" />
   <mkdir dir="${build.dir}" />
   <echo message="creating: ${release.dir}" />
   <mkdir dir="${release.dir}" />
</target>
```

- Alte Dateien löschen

```
<target name="delete">
   <delete dir="${build.dir}" />
</target>
```

3. Übersetzen

- Compilieren von Klassen und Testklassen

```
<target name="compile" depends="prepare">
   <javac srcdir="${source.dir}" destdir="${build.dir}">
     <classpath>
        <path refid="project.classpath" />
     </classpath>
   </javac>

   <javac srcdir="${test.dir}" destdir="${build.dir}">
      <classpath>
         <path refid="project.classpath" />
         <path refid="test.classpath" />
      </classpath>
   </javac>
</target>
```

4. Testen

- JUnit 5-Launcher konfigurieren

```
<target name="test.junit.launcher" depends="compile">
   <junitlauncher haltOnFailure="true" printSummary="true">
      <classpath>
         <path refid="project.classpath" />
         <path refid="test.classpath"/>
      </classpath>
      <testclasses outputdir="${basedir}/${report.dir}/junit">
         <fileset dir="${build.dir}">
            <include name="**/*Test.class" />
         </fileset>
         <listener type="legacy-xml" sendSysOut="true“
                   sendSysErr="true" />
         <listener type="legacy-plain" sendSysOut ="true" />
      </testclasses>
   </junitlauncher>
</target>
```

- Test über Console-Launcher starten und Reporting

```
<target name="test.console.launcher" depends="compile">
   <java classname="org.junit.platform.console.ConsoleLauncher"
         fork="true" failonerror="true">
      <classpath>
         <path refid="project.classpath" />
         <path refid="test.classpath" />
      </classpath>
      <arg value="--scan-classpath" />
      <arg value="--reports-dir ${report.dir}/junit"/>
   </java>
   <junitreport todir="${report.dir}/junit">
      <fileset dir="${report.dir}/junit">
         <include name="TEST-*.xml" />
      </fileset>
      <report format="frames" todir="${report.dir}/junit"/>
   </junitreport>
</target>
```

- Beispiel für ein Testprotokoll

Home

Packages

de.swtd

de.swtd

Classes

BasisTest

Unit Test Results.

Designed for use with JUnit and Ant.

Class de.swtd.BasisTest

Name	Tests	Errors	Failures	Skipped	Time(s)	Time Stamp	Host
BasisTest	2	0	1	0	0.058	2013-12-14T14:45:17	localhost

Tests

Name	Status	Type	Time(s)
testGetter	Success		0.002
testAdd	Failure	expected:<2> but was:<-2> junit.framework.AssertionFailedError: expected:<2> but was:<-2> at de.swtd.BasisTest.testAdd(BasisTest.java:18) at org.eclipse.ant.internal.launching.remote.EclipseDefaultExecutor.executeTargets(EclipseDefaultExecutor.java:32) at org.eclipse.ant.internal.launching.remote.InternalAntRunner.run(InternalAntRunner.java:424) at org.eclipse.ant.internal.launching.remote.InternalAntRunner.main(InternalAntRunner.java:138)	0.010

Properties »

- Testaktivitäten

```
<target name="test" depends="test.junit.launcher, test.console.launcher" />
```

  - Abbrechen des Build-Prozesses bei Fehlern im Unit-Test

```
<fail if="tests.failed" >
</fail>
```

Task ist Ergänzung für Target „test"

4. Ausliefern

- Erzeugen einer `jar`-Datei

```
<target name="deploy" depends="test, compile">
   <jar destfile="${release.dir}/${appname}.jar">
      <fileset dir="${build.dir}" excludes="**/*Test.class" />
         <manifest>
            <attribute name="Main-Class" value="${mainclass}" />
         </manifest>
   </jar>
</target>
```