Simulation und Bildanalyse mit Java
Tests in Informatik und Statistik
Modul-Tests mit JUnit

Robert Pintarelli*

10.05.2004

Zusammenfassung

Durch das systematische Testen von Software zur Entwicklungszeit wird versucht, mögliche Fehler im Programm bereits bei der Erstellung zu erkennen, bzw. zu verhindern. Durch die Automatisierung des Testens ist es möglich effektiv und kostengünstig weniger fehlerbehaftete Software zu erstellen.

Ein Testansatz stellen die sogenannten Unit-Tests dar, die im Umfeld des eXtreme Programming (XP) eingeführt wurden. Unit-Tests zeichnen sich durch ihre einfache Bauart und hohe Lokalität aus. Es werden möglichst kleine Softwareeinheiten (Units) auf Korrektheit getestet. In der OO-Welt dienen oft die einzelnen Klassen als Units.

Eine bei Unit-Tests weit verbreitete Methode des Code-Erzeugens ist die „Test-First“ Methode: Wird eine bestimmte Funktionalität benötigt, wird erst der Test dazu entworfen, dann der eigentliche Programmcode. Der Test gibt dabei die Implementierung mehr oder weniger vor.

JUnit stellt einen Framework dar, um unter Java Unit-Tests auf einfache Weise zu implementieren. Der gut strukturierte Aufbau von JUnit ermöglicht es, den Test-Framework den eigenen Bedürfnissen auf einfache Art und Weise anzupassen.

*robert@pintarelli.name

Inhaltsverzeichnis

Tabellenverzeichnis

Abbildungsverzeichnis

1 Motivation

Software spielt mittlerweile in vielen Bereichen der Wirtschaft aber auch im privaten Haushalt eine grosse Rolle. Vom einfachen MP3-Player bis hin zu hochkomplexen Industrieanlagen werden Softwarepakete eingesetzt, um alle Arten von Aufgaben zu steuern. Die Grösse dieser Softwarepakete kann dabei enorm sein (siehe Tabelle 1). Folie 3

UNIX System V (Release 4.0 inkl. X11)	3.7 Mio. Zeilen Quellcode
R/3 von SAP	7 Mio. Zeilen Quellcode
Windows 2000	30 Mio. Zeilen Quellcode

Tabelle 1: Umfang einiger Software-Systeme

Leider steigt mit dem Umfang eines Software-Pakets auch die Anzahl der Fehler im Programm (siehe Tabelle 2). Folie 4

- 50-60 gefunden Fehler pro 1000 LOC während der Entwickung
- Bis zu 3 Fehler pro 1000 LOC nach der Auslieferung
- Bei sicherheitskritischen Anwendungen ca. 0.5 Fehler pro 1000 LOC

Tabelle 2: Durchschnittliche Fehler in der Software

Einige Beispiele für Softwarefehler finden sich in Tabelle 3. Folie 5

- OB Wahl in Neu-Ulm 1995: 104% Wahlbeteiligung (tatsächlich 52%)
- Bluescreen
- Gepäckverteilungsanlage Flughafen von Denver: 16 Monate verzögerung, Schaden 550Mio USD
- Golfkrieg 1991: 28 Tote, fast 100 Verletzte wg. fehlerhafter Uhrsynchronisation

Tabelle 3: Beispiele für Fehler in Software

Die durchschnittliche Fehlerquote bei Programmen hat zum sogenannten *zweiten* Weinberg*'schen Gesetz* geführt: Folie 6

> Wenn Architekten ihre Gebäude so bauen würden, wie Programmierer ihre Programme schreiben, würde ein einziger Specht die ganze Zivilisation zerstören.[1]

Obwohl Software ein Industriegut darstellt, zeigen sich doch deutliche Unterschiede zu traditionellen Gütern: Folie 7

- Software ist immateriell
- Software ist komplex und zeigt keine „natürlichen“ Strukturen
- Software hat keine natürliche „Lokalität“ (Fehlereingrenzung dadurch oft sehr schwierig)
- Software ist nicht stetig

Das oben Gesagte zeigt, das Fehler in Software aufgrund des Grösse der Softwarepakete praktisch unvermeidbar sind. Dennoch können Fehler erhebliche Folgen nach sich ziehen, nicht nur materielle, sondern auch rechtliche. Ein grosser Aspekt bei der Softwareerstellung sollte daher das frühzeitige Erkennen und Beseitigen von Fehlern sein.

2 Unit-Tests

Unit- oder auch Modul-Tests zeichnen sich durch ihre einfache Bauart und hohe Lokalität aus. Ziel dieser Tests ist es, möglichst einfache Bausteine einer komplexen Softwarestruktur auf Korrektheit zu testen bevor diese in ein Softwareprodukt eingehen. Ein wichtiger Aspekt hierbei ist die Unabhängigkeit der Testfälle untereinander. Das Ergebnis eines Tests darf das Ergebnis eines anderen Tests nicht beeinflussen. Dieses Ziel ist natürlich nicht erreichbar, denn sobald Software komplexer wird, nehmen auch die Abhängigkeiten zwischen den einzelnen Moduln der Software zu. Unit-Tests führen aber dazu, das bereits bei der Softwareentwicklung darauf geachtet wird, diese Abhängigkeiten möglichst gering und einfach zu halten, so dass im Idealfall die einzelnen Bauteile eines Softwareprojekts einzeln testbar (und somit auch austauschbar) sind. Dies führt auch dazu, dass frühzeitig mit dem Testen begonnen werden kann, da die Module unabhängig sind. Folie 8

2.1 OO und Unit-Tests

Besonders in der Objektorientierten Welt sind Unit-Tests sehr beliebt. Die einzelnen Klassen einer OO-Architektur bieten sich als Units geradezu an. Denn auch beim OO-Design wird oft nach dem „Keep-it-simple“ Prinzip, sowie dem Prinzip der Wiederverwertbarkeit gearbeitet. Das sind beides Eigenschaften die auch den Unit-Tests zugrunde liegen bzw. durch diese gefördert werden.

2.2 Anforderungen an einen Unit-Test Framework

Das Durchführen von Tests erfordert den Einsatz von geeigneten Werkzeugen, die den Testprozess aktiv unterstützen. Diese Werkzeuge sollten den folgenden Anforderungen genügen (siehe auch [2]): Folie 9

1. Die Sprache zur Testspezifikation ist die Programmiersprache selbst
2. Anwendungscode und Testcode müssen getrennt werden können
3. Die Ausführung und Verifikation einzelner Testfälle ist voneinander unabhängig
4. Testfälle können beliebig in Testsuiten zusammengefasst werden
5. Der Erfolg oder Misserfolg der Testausführung muss auf einen Blick erkennbar sein.

zu 1) Der Einsatz der selben Programmiersprache erleichtert das Schreiben von Testfällen. Der Programmierer muss erstens keine weitere Sprache erlernen und zweitens muss nicht immer „umdenken“ wenn er einen Test entwirft.[1]

zu 2) Dies dient zum einen der Übersichtlichkeit, zum anderen ist es auch gar nicht notwendig, den Testcode auszuliefern (toter code).

zu 3) Entscheidend ist, dass die Ausführung eines Tests keine Auswirkungen auf nachfolgende Testfälle hat. Andernfalls führen Abhängigkeiten zwischen Tests zu nicht lokalen Auswirkungen einzelner Fehler.

zu 4) Oft will man aus Zeitgründen nicht alle Tests durchführen. Deshalb ist das Gruppieren von Tests wichtig. Da diese Zusammenfassungen verschiedene Schwerpunkte setzen können, ist es wichtig, dass die Testfälle beliebig kombinierbar sind.

zu 5) Wer schon mal die Fehlerausgabe des GNU C++ Compilers gesehen hat, weiss was eine kurze Fehlermeldung wert ist.

3 JUnit

JUnit[4] stellt einen Framework zum Druchführen von Unit-Tests unter Java[5] dar. Im Folgenden wird die Version 3.8.1 von JUnit, sowie die Version 1.4.1 von Java verwendet. JUnit selbst ist ebenfalls in Java verfasst(1). Folie10

[1] Eine eigene Testsprache kann aber auch eine Erleichterung darstellen, siehe auch [3]

3.1 JUnit im Überblick

Bei JUnit dreht sich alles mehr oder weniger um die Klasse `TestCase`. Diese Klasse stellt den Kern einer jeden Test-Unit dar. Jede Klasse die Testfälle beinhaltet muss von `TestCase` abgeleitet sein(2). Innerhalb dieser Klasse können einzelne Testsfälle implementiert werden. Die Methoden hierzu sollten alle mit dem Wort *test* beginnen. Ausserdem wird über die Methode *suite* die Möglichkeit geboten, Testfälle zu kombinieren(3). Weitere Methoden zum Beeinflussen der einzelnen Testfälle stellen die Methoden *setUp()* sowie *tearDown()* dar(4).

Die Testfälle können dann automatisiert durchgeführt werden; dazu stehen drei Möglichkeiten zur Verfügung: Folie11

junit.textui.TestRunner textuelle kompakte Darstellung der Testergebnisse

junit.awtui.TestRunner grafische Darstellung mit Hilfe der AWT

junit.swingui.TestRunner grafische Darstellung mit Hilfe von Swing

Alle drei Methoden liefern das selbe Ergebnis, sie können je nach Geschmack benutzt werden(5).

JUnit erfüllt durch (1)-(5) die in Kaptitel 2.2 gestellten Anforderungen an einen Testframework.

4 Unit-Tests mit JUnit

Der Softwareentwicklungsprozess mit Hilfe von JUnit wird an folgendem Beispiel gezeigt:

Es soll eine Erweiterung zu einem bestehenden Projekt erstellt werden. Dabei handelt es sich um eine Accountverwaltung. Es soll die Möglichkeit geschaffen werden, jedem Account ein oder mehrere Rechner zuzuordnen. Für statistische Zwecke soll ebenfalls verwaltet werden, welche Betriebssysteme auf den jeweiligen Rechnern zum Einsatz kommen, und wer der Hersteller des jeweiligen Betriebssystems ist. Da das Softwareprodukt auch zum Verwalten eines DHCP-Servers[6] benutzt wird, müssen zusätzlich zum DNS-Namen des Rechners[7] auch noch seine IP im dotted-decimal Format[8], sowie seine MAC-Addresse[9] verwaltet werden. Ausserdem muss die Möglichkeit vorhanden sein, einzustellen ob ein Rechner das Netzwerk überhaupt benutzen darf. Es darf jeweils nur einen aktiven Rechner pro MAC/IP Paar geben. Alle Vorgaben spiegeln sich in Abbildung 1 wider.

Für das Projekt gibt es zwei Packages: `seminar` und `test`. `seminar` enthält den Produktivcode, `test` die Testfälle.

4.1 Der Test-First Ansatz

Das Projekt wird mit Hilfe des „Test-First“ Ansatzes realisiert. Ziel dieses Ansatzes ist es, mit Hilfe von Testfällen die Implemtierung vorzugeben (Test-Driven Developement). Es wird (nur) solange Code erzeugt, bis alle Tests erfolgreich sind. Da die Vorgaben im Beispiel sehr exakt sind, ist es einfach die Testfälle zu formulieren. Folie12

Beim „Test-First“ Ansatz wird oft die „Bottom-Up“-Methode zum Entwickeln der Klassenhierarchie benutzt. Als erstes werden die Klassen implementiert, die am wenigsten Abhängikeiten zu anderen Klassen haben, und am „weitesten unten“ im Klassendiagramm sind.

Bei dieser Methode kann es häufiger vorkommen, dass es Aufgrund der Testfälle zu einer Reorganisation des Klassencodes und der Klassenhierarchie kommt.

4.2 Generieren von Testfällen

Das Generieren von Testfällen unter JUnit ist sehr einfach: Eine Klasse mit Tests muss lediglich `junit.framework.TestCase` erweitern, und Methoden die die Tests durchführen bereitstellen. Eine Konvention hierbei ist, dass die Methoden einheitlich mit *test* beginnen[2]. Die Methoden *assertTrue(boolean)* und *assertFalse(boolean)* bieten die Möglichkeit, Ergebnisse von Aufrufen zu testen. Der Rumpf für eine Testklasse kann also folgendermassen aussehen: Folie13

```
import junit.framework.*; //JUnit bekannt machen

public class MeinTest extends TestCase {
  public MeinTest(String name) { //muss so sein
    super(name);
  }
  public void testTrivial() { //ein test
    assertTrue(true);
  }
}
```

4.3 simpleOS und simpleOSTest

Die Betriebssysteminformationen haben keine Abhängigkeiten zu anderen Folie14

[2]Dies sorgt auf dafür, dass JUnit mit Hilfe der Java Reflection API die auszührenden Methoden findet

Klassen, deshalb wird diese Klasse als erstes Implementiert. Um den „Test-First“ Ansatz gerecht zu werden, müssen erst die Testfälle formuliert werden. Dies geschieht in der Klasse `test.simpleOSTest`: *step1*

```
public class simpleOSTest extends TestCase {

    public simpleOSTest(String name) {
        super(name);
    }

    public void testGetName() {
        simpleOS _os = new simpleOS("Linux", "open source");
        assertTrue(_os.getName().equals("Linux"));
    }

    public void testGetCompany() {
        simpleOS _os = new simpleOS("Linux", "open source");
        assertTrue(_os.getCompany().equals("open source"));
    }

}
```

Durch die Testfälle wird hier direkt die Implementierung von `seminar.simpleOS` vorgegeben:

```
public class simpleOS {
    private String _name;
    private String _company;

    public simpleOS(String name, String company) {
        _name = name;
        _company = company;
    }

    public String getName() {
        return _name;
    }

    public String getCompany() {
        return _company;
    }
}
```

4.3.1 der erste Testlauf

Nach dem Implementieren der Klassen erfolgt der erste Testlauf:

```
java -cp ./build:/usr/share/java/junit.jar \
junit.textui.TestRunner test.simpleOSTest
..
Time: 0.022

OK (2 tests)
```

Wie zu erwarten war, sind beide Tests erfolgreich verlaufen. Ein „.“ in der Ausgabe steht hierbei für einen erfolgreich absolvierten Test.

4.4 Erweiterung von simpleOS

Um die Performance des Systems zu gewährleisten wird vorgebeben, dass im fertigen Produkt jeder Rechner ein `HashSet` seiner zugehörigen Betriebssysteme verwaltet. Diese Vorgabe erzwingt aber, dass `simpleOS` die Methoden *hashCode()* und als Konsequenz daraus auch die Methode *equals()* bereitstellen muss[10].

4.4.1 die equals() Methode

In der Standardimplementierung der Methode *equals()* in `java.lang.Object` findet eine Identitätsprüfung statt, d.h. es wird nur dann *true* zurückgeliefert, wenn die beiden Objekte den gleichen Speicher referenzieren. Dieses Verhalten ist hier nicht erwünscht, deshalb wird *equals()* in `simpleOS` neu implementiert.

Dabei ist zu beachten, dass JAVA folgende Eigenschaften von der *equals()*-Methode fordert[11]: Folie15

- Sie muss transitiv sein.
- Sie muss reflexiv sein.
- Sie muss symmetrisch sein.
- Ein Vergleich mit *null* muss immer *false* ergeben.
- Nur der Vergleich mit Objekten gleichen Inhalts[3] und gleichen Typs darf *true* ergeben.

[3]Was genau den zu vergleichende Inhalt darstellt, liegt im ermessen des Programmierers.

Diese Anforderungen führen also zur Erweiterung um folgende Testfälle: Folie16

- *public void testEqualsSymmetric();*
- *public void testEqualsReflexiv();*
- *public void testEqualsTransitiv();*
- *public void testEqualsSame();*
- *public void testEqualsNot();*
- *public void testEqualsOther();*
- *public void testEqualsNull();*

Auffällig hierbei ist, das für jeden Test, um die Unabhängigkeit der Tests zu gewährleisten, ein neues Objekt von Typ `simpleOS` benötigt wird.

Fixtures Um jedem Testfall die gleiche Umgebung zu schaffen, bietet JUNIT die Möglichkeit, die Methode *setUp()* zu überschreiben. In Ihr können Folie17 Aktionen definiert werden, die vor jedem Aufruf einer Testmethode durchgeführt werden sollen. Im den obigen Tests ist die also das Bereitstellen neuer Objekte in den Klassenvariablen *_os, _os2, _os3, _os4*:

```
public void setUp() throws Exception {
    super.setUp();
    _os = new simpleOS("Linux", "open source");
    _os2 = new simpleOS("Linux", "open source");
    _os3 = new simpleOS("Linux", "open source");
    _os4 = new simpleOS("Windows", "open source");
}
```

Analog zu *setUp()* kann über die Methode *tearDown()* Code hinterlegt werden, der nach jedem Test ausgeführt werden soll.
setUp() und *tearDown()* werden immer ausgeführt.

4.4.2 Implementation der equals()-Methode

Zunächst wird die Methode nur wie folgt Implementiert: *step2*

```
public boolean equals(Object other) {
    return false;
}
```

Diese Implementation ergibt erwartungsgemäss die folgenden Testergebnisse:

```
java -cp ./build:/usr/share/java/junit.jar \
junit.textui.TestRunner test.simpleOSTest
...F.F.F.F...
Time: 0.028
There were 4 failures:
1) testEqualsSymmetric(test.simpleOSTest)...
...
2) testEqualsReflexiv(test.simpleOSTest)...
...
3) testEqualsTransitiv(test.simpleOSTest)...
...
4) testEqualsSame(test.simpleOSTest)
...
FAILURES!!!
Tests run: 9,  Failures: 4,  Errors: 0
```

Der „Test-First“ Ansatz schreibt jetzt vor, dass die Methode so erweitert wird, dass die vier Fehlgeschlagenen Tests beim nächsten Lauf erfolgreich absolviert werden können. Dies führt zu folgender Implementation: *step3*

```
public boolean equals(Object other) {
    if (this == other) {
        return true;
    }
    if (other instanceof simpleOS) {
        simpleOS otherOS = (simpleOS)other;
        if (getName().equals(otherOS.getName())) {
            return true;
        }
    }
    return false;
}
```

Ein erneuter Testflauf führt jetzt zum gewünschten Erfolg.

4.4.3 die hashCode()-Methode

Für die Methode *hashCode()* ist nur die Eigenschaft vorgeschrieben, das Ojekte die bei Vergleichen mit *equals()* gleich sind, auch den den selben Hash-Wert Folie18

liefern[11]. Der folgende Testfall bildet diese Anforderung ab: *step4*

```
public void testHashCode() {
    assertTrue(_os.hashCode() == _os2.hashCode());
}
```

In der Klasse `simpleOS` ist die Methode folgedermassen umgesetzt: *step5*

```
public int hashCode() {
    return getName().hashCode();
}
```

4.5 die simpleHost Klasse und ihre Helfer

Da die Klasse `simpleOS` jetzt allen Anforderungen genügt, kann die Klasse `simpleHost` implementiert werden. Zunächst wird das Hinzufügen eines Betriebssystems zu einem Rechner getestet. Dazu werden `test.simpleHost Test` und `seminar.simpleHost` wie folgt implementiert: *step6*

```
public class simpleHostTest extends TestCase {
    private HashSet _oss;
    private simpleHost _host;
    private simpleOS _os1;
    private simpleOS _os2;

    public simpleHostTest(String name) {
        super(name);
    }

    public void setUp() throws Exception {
      super.setUp();
      _oss = new HashSet();
      _host = new simpleHost();
      _os1 = new simpleOS("Linux", "open source");
      _os2 = new simpleOS("Windows", "Microsoft Corporation");
    }

    public void testAddOs() {
        _host.addOS(_os1);
        _oss.add(_os1);
        assertTrue(_oss.equals(_host.getOSs()));
    }
```

```
    public void testAddTwoOs() {
        _host.addOS(_os1);
        _host.addOS(_os2);
        _oss.add(_os1);
        _oss.add(_os2);
        assertTrue(_oss.equals(_host.getOSs()));
    }

    public void testAddDuplicateOs() {
        _host.addOS(_os1);
        _host.addOS(_os1);
        _oss.add(_os1);
        assertTrue(_oss.equals(_host.getOSs()));
    }
};


public class simpleHost {
...
    /** Betriebssysteme */
    private HashSet _oss;
...
    public HashSet getOSs() {
        return new HashSet(_oss);
    }

    public void addOS(simpleOS os) {
        _oss.add(os);
    }
...
}
```

Der Test der neuen Funktionlität ergibt folgendes Ergebnis:

```
java -cp ./build:/usr/share/java/junit.jar \
junit.textui.TestRunner test.simpleHostTest
.E.E.E
Time: 0.033
There were 3 errors:
1) testAddOs(test.simpleHostTest)
   java.lang.NullPointerException
```

```
2) testAddTwoOs(test.simpleHostTest)
   java.lang.NullPointerException
3) testAddDuplicateOs(test.simpleHostTest)
   java.lang.NullPointerException

FAILURES!!!
Tests run: 3,  Failures: 0,  Errors: 3
```

Offensichtlich funktioniert das Hinzufügen eines Betriebssystems nicht erwartungsgemäß. Der Grund hierfür ist das Fehlen eines Konstruktors in `simpleHost`, der das `HashSet` *_oss* korrekt initialisiert. Das Hinzufügen folgenden Codes zu `simpleHost` löst das Problem: *step7*

```
public simpleHost() {
    _oss = new HashSet();
};
```

Der erneute Testlauf liefert jetzt dieses Ergebnis:

```
java -cp ./build:/usr/share/java/junit.jar \
junit.textui.TestRunner test.simpleHostTest
...
Time: 0.028

OK (3 tests)
```

Da ein Rechner einen DNS-Konformen Namen, seine IP im dotted-decimal Format und der dazugehörigen MAC-Addresse besitzen soll, werden weitere Tests benötigt. Die Vorgaben, wie die genannten Attribute gestaltet sein müssen, finden sich in [7], [8] und [9]. Das Testen dieser Eigenschaften ist aber nicht direkt Aufgabe der Klasse `simpleHost`, deshalb werden sie in die Klassen Folie19

1. `seminar.RFC1034`
2. `seminar.RFC1117`
3. `seminar.ieee802`

ausgegliedert, und entsprechend getrennt getestet.

4.5.1 Testsuiten

Um so mehr verschiedene Testklassen vorhanden sind, desto aufwändiger wird das Ausführen der einzelnen Testfälle. Daher können die Testklassen zu einer `junit.framework.TestSuite` zusammengefasst werden. Mit Hilfe der *addTestSuite()* Methode ist es möglich, einzelne Testfälle, aber auch ganze Testklassen mit in die TestSuite aufzunehmen. Folie20

Gängige Praxis ist es, eine Klasse `AllTests` zu erstellen, die alle Tests beinhaltet. Im Beispiel hat diese Klasse also folgenden Implementation: *step9*

```
public class AllTests  {
    public static Test suite() {
        TestSuite suite = new
                  TestSuite("Test suite fürs seminar");
        suite.addTestSuite(test.simpleOSTest.class);
        suite.addTestSuite(test.simpleHostTest.class);
        suite.addTestSuite(test.RFC1034Test.class);
        suite.addTestSuite(test.RFC1117Test.class);
        suite.addTestSuite(test.ieee802Test.class);
        return suite;
    }

    public static void main(String[] args) {
        junit.textui.TestRunner.run(test.AllTests.suite());
    }
}
```

Alle Tests können jetzt also folgendermassen ausgeführt werden:

```
java -cp ./build:/usr/share/java/junit.jar test.AllTests
.........................
Time: 0.166

OK (25 tests)
```

4.5.2 simpleHost Implementation

Da jetzt alle Vorraussetzungen für `simpleHost` geschaffen sind, können die Klasse und die entsprechenden Testfällen implementiert werden: Folie21 *step10*

```
public class simpleHostTest extends TestCase {
...
    public void testGetId() {
```

```
        _host.setId(1);
        assertTrue(_host.getId() == 1);
    }
    public void testGetMAC() {
        _host.setMAC("00:00:e2:2f:69:77");
        assertTrue(_host.getMAC().equals("00:00:e2:2f:69:77"));
    }
    public void testGetName() {
        _host.setName("theseus");
        assertTrue(_host.getName().equals("theseus"));
    }
    public void testGetIP() {
        _host.setIP("136.60.166.2");
        assertTrue(_host.getIP().equals("136.60.166.2"));
    }
    public void testGetActive() {
        _host.setActive(true);
        assertTrue(_host.isActive());
    }
...
}

public class simpleHost {
    private int _id;
    private String _mac;
    private String _name;
    private String _ip;
    private boolean _active = false;
...
    public simpleHost() {
        _id = -1;
        _mac = "";
        _name = "";
        _ip  = "";
        _active = false;
        _oss = new HashSet();
    };
...
    public void setId(int id) {
        _id = id;
    }
```

```
public void setMAC(String mac) {
    if (!ieee802.checkMAC(mac)) {
        throw(new IllegalArgumentException(
              mac + " ist nicht korrekt!"));
    }
    _mac = mac;
}

public void setName(String name) {
    if (!RFC1034.checkName(name)) {
        throw(new IllegalArgumentException(
              name + " ist nicht korrekt!"));
    }
    _name = name;
}

public void setIP(String ip) {
    try {
        if (!RFC1117.checkDottedIP(ip)) {
            throw(new IllegalArgumentException(
                  ip +" ist nicht korrekt!"));
        }
    } catch (Exception e) {
        throw(new IllegalArgumentException(
              ip + "Fehler:"+e.getMessage()));
    }

    _ip = ip;
}

public void setActive(boolean active) {
    _active = active;
}

public int getId() {
    return _id;
}

public String getMAC() {
    return _mac;
```

```
    }

    public String getName() {
        return _name;
    }

    public String getIP() {
        return _ip;
    }

    public boolean isActive() {
        return _active;
    }
}
```

Ein abschliessender Test zeigt, dass alle Testfälle erfolgreich verlaufen:

```
java -cp ./build:/usr/share/java/junit.jar test.AllTests
.............................
Time: 0.173

OK (30 tests)
```

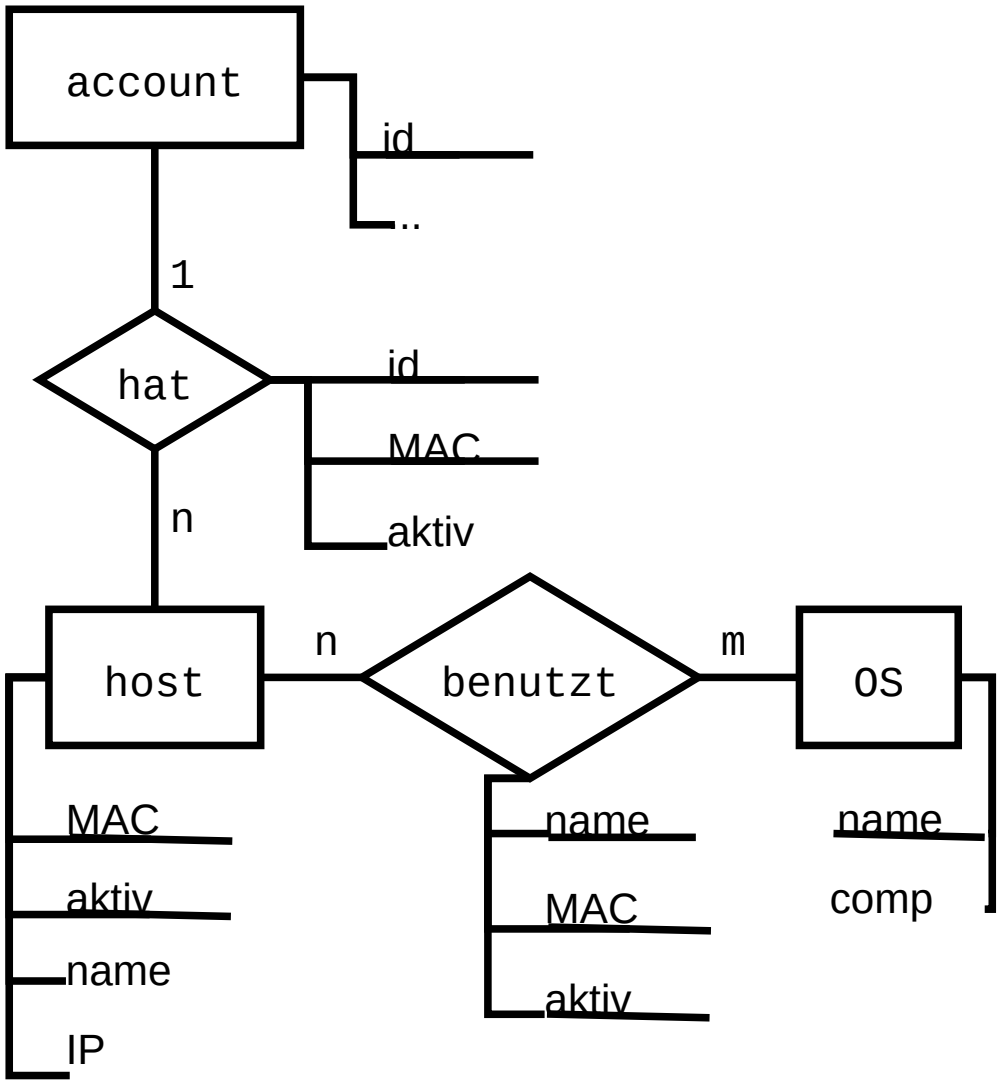

Abbildung 1: Beispiel ER-Diagramm

5 Das Innenleben von JUnit

Der gut strukturierte Aufbau von JUnit ermöglicht es, den Framework sehr einfach den eigenen Bedürfnissen anzupassen. Um eigene Erweiterungen in JUnit zu integrieren ist es jedoch erforderlich, sich mit dem internen Aufbau von JUnit zu befassen.

5.1 Der interne Aufbau von JUnit

JUnit wurde konsequent mit Hilfe sog. Entwurfsmuster (engl. Design Pat terns[12]) entwickelt. Im Folgenden werden einige Klassen vorgestellt, die das Grundgerüst von JUnit bilden.

Die Testfälle werden durch die Klasse `TestCase` repräsentiert, sie ist als Command-Pattern implementiert. In [12] wird die Aufgabe dieses Musters wie folgt beschrieben: Folie22

> Encapsulate a request as an object, thereby letting you parameterize clients with different requests...

Es soll also eine Aufgabe in ein Objekt gekapselt werden, um gewisse „Dinge“ auf einstellbare Art zu erledigen. `TestCase` definiert hierfür die Methode *run()*.

Da alle Tests im wesentlichen den selben Ablauf haben, nämlich das Anlegen der Testumgebung (s. Fixtures 4.4.1), das Ausführen eines Testfalls und das „Aufräumen“ der Umgebung, bietet `TestCase` hierfür Schablonen (engl. Template Methods) an.

- *setUp()*
- *runTest()*
- *tearDown()*

Diese drei Methoden werden von *run()* nacheinander aufgerufen. *setUp()* und *tearDown()* sollten in eigenen Testfällen überschrieben werden, um die den gewünschten Kontext herzustellen.

Um eine Statistik über die Fehler zu generieren, werden alle Testfälle über die Klasse `TestResult` mitprotokolliert. Sie ist in Form eines Zählparameters (Collecting Parameter) implementiert, der *run()* übergeben wird. Folie23
`TestResult` enthält Zähler und Methoden, um die Ergebnisse der einzelnen Testfälle, wie Erfolg, Fehler, Aushnahmen, usw. zu erfassen. Damit *run()* Fehlschläge als solche erkennen, und sie an `TestResult` weiterleiten kann, müssen die einzelnen Testfälle eine Nachricht an *run()* senden, wie der Test

verlaufen ist. Hierfür dienen die Methoden *assertEquals(), assertTrue()* und *assertFalse()*. Im Fehlerfall lösen sie eine Ausnahme (Exception) aus, die von *run()* ausgewertet wird. Falls keine Ausnahme auftritt, wird dies als erfolgreicher Testlauf interpretiert.

Da *run()* nur *runTest()* zum Ausführen eines Testfalls benutzt, muss entweder die Methode *runTest()* für jeden Testfall individuell überschrieben werden, oder aber *runTest()* wird darüber in Kenntnis gesetzt, wie die richtige Testmethode heisst. JUnit bedient sich dazu der „Java Reflection API“. Folie24
runTest() erhält über eine Klassenvariable den Namen der auszuführenden Funktion, und generiert daraus einen Methodenaufruf. Die Klassenvariable wird durch den Konstruktor der Klasse gesetzt. [4].

`TestCase` ist jetzt in der Lage, über *run()* einen Testfall auszuwerten, der Folie25
von *runTest()* ausgeführt wird. Um transparent mehrere Tests in einem Lauf durchführen zu können, sind weitere Schritte notwendig.

In JUnit wird dafür ein Verbund eingesetzt.

Verbund: Ein Verbund (engl. Composite) besteht im Idealfall aus einer den Verbund nach aussen hin repräsentierenden abstrakten Klasse, sowie aus seinen Mitgliedern, die diese Klasse erweitern. Da die abstrakte Klasse rein abstrakt ist, kann in Java auch ein Interface benutzt werden.

Ein Verbund verbindet also verschiedenartige Klassen über ein Interface.

In JUnit ist dieser Verbund wie folgt realisiert: Folie26

1. `Test` stellt das Interface dar, das die Mitglieder implementieren.
2. `Test` definiert die Methode *run()* um einen Test zu starten
3. `TestCase` ist ein Mitglied des Verbunds. Diese Klasse erfüllt schon alle Eigenschaften um Mitglied zu sein.
4. `TestSuite` wird als Mitglied neu eingeführt. Sie enthält eine Liste von `TestCase`-Objekten, und Methoden um diese Liste zu Pflegen. Die *run()*-Methode iteriert über diese Liste und ruft die jeweiligen *run()*-Methoden der Listenelemente auf.

Abbildung 2 zeigt den geschilderten Aufbau. Folie27

Über *Test.run()* können also eine komplette TestSuites, aber auch nur einzelne Testfälle abgearbeitet werden. Das Ausführen dieser `Test`-Klassen übernehmen in JUnit die `junit.*.TestRunner` Klassen.

[4] deshalb lautet die Signatur des Konstruktors auch *KlassenName(String name)*

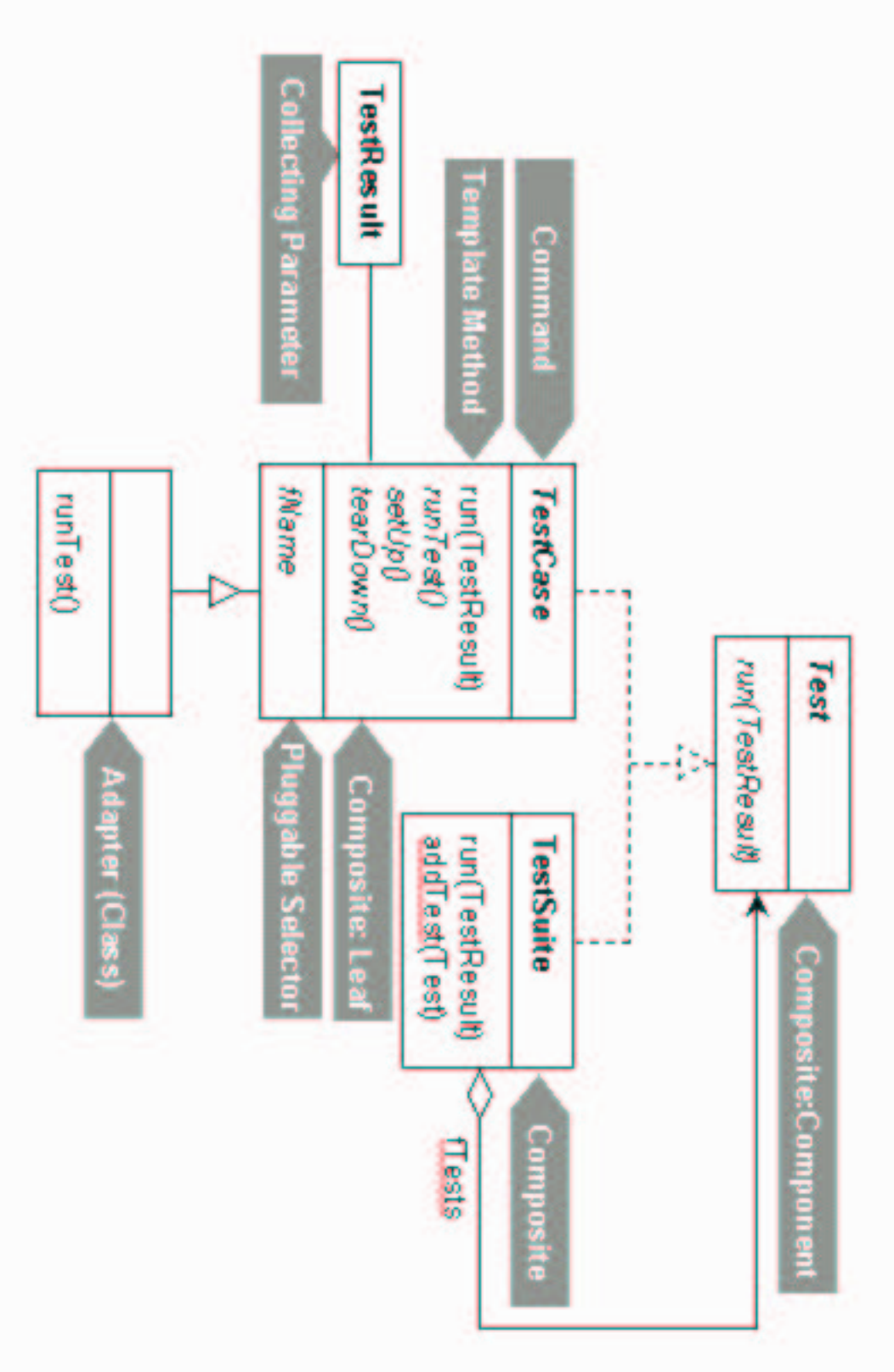

Abbildung 2: JUnit Klassendiagramm

5.2 DBUnit als Erweiterung von JUnit

Das Beispiel in Kapitel 4 verwendet als Grundlage ein ER-Diagramm, die Klassen `simpleHost` und `simpleOS` haben aber bisher keinerlei Datenbankfunktionalität. JUnit bietet von sich aus auch keine Methoden, um Datenbankapplikationen zu testen. Diese Lücke füllt DBUnit[13] aus. Es erweitert die Klasse `TestCase` zu `DatabaseTestCase`. In dieser erweiterterten Klasse Folie28
sind zwei weitere (abstrakte) Schablonen definiert:

- *getConnection()*
- *getDataSet()*

getConnection() soll die Datenbankverbindung aufbauen, während *getDataSet()* die Datenbank mit definierten Werten füllt. *getDataSet()* ist also vergleichbar mit der *setUp()*-Methode.

Ausserdem erweitert DBUnit die *assertXXX()*-Methoden um Funktionen, die den Inhalt von Tabellen mit einem Sollzustand vergleichen.

Das Package `seminar` enthält zum Schreiben von Objekten in eine Datenbank die Klasse `DBWriter`. Die Methode zum schreiben eines Betriebssystems hat die folgenden Implementation: Folie29

```
public void writeObject(Connection con, simpleOS os) {
    PreparedStatement ps =
        con.prepareStatement("insert into os values(?,?)");
        ps.setString(1, os.getName());
        ps.setString(2, os.getCompany());
        ps.execute();
    }
```

Der Test[5] ist etwas aufwändiger:

```
public void testWriteDB() throws Exception {
  Connection con = getTestConnection();
  _writer.writeObject(con, _os);

  // Fetch database data after executing your code
  IDataSet databaseDataSet = getConnection().createDataSet();
  ITable actualTable = databaseDataSet.getTable("OS");

  // Load expected data from an XML dataset
```

[5] hier ist der Übersicht wegen nur die Testmethode aufgeführt

```
  IDataSet expectedDataSet =
    new FlatXmlDataSet(
        new FileInputStream("test/full-os-correct.xml"));
  ITable expectedTable = expectedDataSet.getTable("OS");

  // Assert actual database table match expected table
  Assertion.assertEquals(expectedTable, actualTable);

}
```

Diese Methode versucht, einen Eintrag in die Tabelle zu schreiben, und vergleicht dann diese Tabelle mit den Solldaten, die in „full-os-correct.xml“ als XML-Struktur hinterlegt sind. Nach dem Anpassen der `AllTests` Klasse ergibt sich jetzt folgendes Testergbnis:

```
java -cp ./build:./lib/pg72jdbc2.jar: \
   ./lib/dbunit-2.0.jar:/usr/share/java/junit.jar \
   test.AllTests

.............................
Time: 1,035

OK (31 tests)
```

Literatur

[1] Prof. Helmut Partsch. *Softwaretechnik*. Universiät Ulm.

[2] J. Link. *Unit Tests mit Java*. dpunkt Verlag, 2002.

[3] J.M. Spivey. *The Z Notation: A Reference Manual*.

[4] Erich Gamma et al. *JUnit, Testing Resources for Extreme Programming*. http://www.junit.org/.

[5] Sun Microsystems Inc., http://java.sun.com/. *Java Technology*.

[6] R. Droms. *Dynamic Host Configuration Protocol*. Network Working Group, http://www.ietf.org/rfc/rfc1034.txt.

[7] P. Mockapetris. *DOMAIN NAMES - CONCEPTS AND FACILITIES*. Network Working Group, http://www.ietf.org/rfc/rfc1034.txt.

[8] S. Romano, M. Stahl, and M. Recker. *INTERNET NUMBERS*. Network Working Group, http://www.ietf.org/rfc/rfc1117.txt.

[9] IEEE, http://standards.ieee.org/getieee802/. *IEEE 802.2: Logical Link Control*.

[10] Brian Goetz. *Java theory and practice: Hashing it out*. http://www-106.ibm.com/developerworks/java/library/j-jtp05273.html.

[11] Sun Microsystems Inc., http://java.sun.com/docs/books/. *The Java Language Specification*.

[12] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley Professional Computing Series. Addison-Wesley Publishing Company, New York, NY, 1995.

[13] DBUnit Developement Team. *DBUnit 2.0*. http://www.dbunit.org/.

[14] Erich Gramma. *JUnit A Cook's Tour*. http://junit.sourceforge.net/doc.

[15] *PostgreSQL*. http://www.postgresql.org.