

Testgesteuert Entwickeln mit JUnit 4

JUnit

Tests „nebenbei“ automatisieren

Christoph Knabe
Beuth-Hochschule für Technik Berlin

1. Inhalt

Inhaltsverzeichnis

1. Inhalt.....	2
2. Traditionelle Testverfahren.....	3
3. Klassifizierung von Tests.....	5
4. Test Driven Development (TDD).....	6
5. Beispiel: TDD einer Klasse Stack für Strings.....	7
6. Klasse minimal implementieren:.....	8
7. „last in – first out“ (LIFO) testen.....	9
8. Ausgabe des LIFO-Tests bei Ausführung:.....	10
9. popInvertsPush-Eigenschaft herstellen.....	11
10. Testtreiber ergänzen: Nach 4 push()s und 4 pop()s muss Stack wieder leer sein.....	12
11. emptyAfterPushesAndPops befriedigen:.....	13
12. Testtreiber ergänzen: pop() auf leerem Stack läßt ihn leer.....	14
13. popOnEmptyDoesNotModify befriedigen:.....	15
14. Round Trip Test.....	16
15. Best Practice Quellcodeorganisation.....	17
16. Testtreiber als Lernmittel.....	18
17. Test-Suite zum Ausführen von Testtreibern in definierter Reihenfolge.....	20
18. Fazit.....	21

2. Traditionelle Testverfahren

1. Entwickeln
2. Aufstellen einer Testmatrix mit 3 Spalten
 - zu tätige Vorbereitung/Eingabe
 - erwartete Ausgabe/Reaktion
 - tatsächliche Ausgabe/Reaktion
3. Manuelles Durchführen vor jedem Release
4. Überarbeiten des Quellcodes

¿? Probleme?

Probleme traditioneller Testverfahren

- Tests spät im Phasenmodell ⇒ oft Budget erschöpft
- Spezifische Codeteile wie Fehlerbehandlung schwer über Oberfläche ansteuerbar
⇒ Tests kleinerer Einheiten nötig!
Bsp.: Test von Datenbankschreibern bei voller Platte
- Keine Garantie, dass alle Codeteile getestet werden
⇒ Tests kleinerer Einheiten nötig!
- Manuelle Testdurchführung sehr aufwändig ⇒
 - selten machbar (unagil)
 - Fehler aufgrund Ermüdung

3. Klassifizierung von Tests

- Nach **Größe** des Getesteten:
 - Eine Unit (z.B. Klasse) isoliert: **Unit-Test**
 - Mehrere Units im Zusammenspiel:
Integrationstest
 - Gesamtsystem über Oberfläche: **Akzeptanztest**
- Nach **Testabdeckung** von:
 - Ablaufzweigen
 - Ablaufbedingungen
 - Ablaufpfaden
- Nach **Automation**
- Nach **Wiederholung**: einmalig, Regressionstest

4. Test Driven Development (TDD)

- Eingeführt ab 1998 von **Kent Beck** für Smalltalk.
- Entwicklungszyklus:
 1. Testtreiber programmieren, der gewünschtes Verhalten prüft.
 2. Ausführen Testtreiber+Testling:
Sollte Fehler feststellen.
 3. Testling erweitern/korrigieren bis Fehler weg.
- Vorteile:
 - umfangreiche Testsuite „nebenbei“
 - hohe Testabdeckung
 - Es entstehen nur testbare Units.

5. Beispiel: TDD einer Klasse Stack für Strings

Klasse spezifizieren

```
/**Stack of Strings according to the "Last In - First Out" (LIFO) principle.*/  
public interface Stack {  
  
    /**Puts element onto the stack*/  
    void push(String element);  
  
    /**Returns and removes the oldest element of the stack.  
    @throws EmptyExc No element is on the stack*/  
    String pop() throws EmptyExc;  
  
    /**No element is on the stack*/  
    class EmptyExc extends Exception {}  
  
}
```

6. Klasse minimal implementieren:

```
public class StackImpl implements Stack {  
  
    @Override  
    public void push(final String element) {  
    }  
  
    @Override  
    public String pop() throws EmptyExc {  
        return null;  
    }  
  
}
```

¿? Fehlt hier was?

7. „last in – first out“ (LIFO) testen

```
import static org.junit.Assert.*;
import org.junit.Test;
public class StackTest {

    private final Stack stack = new StackImpl();

    @Test public void popInvertsPush() throws Exception {
        final String[] elements = {"", "a", "XY", "749398"};
        //Push all:
        for(final String string: elements){
            stack.push(string);
        }
        //Pop all and compare:
        for(int i=elements.length-1; i>=0; i--){
            assertEquals(elements[i], stack.pop());
        }
    }
}
```

8. Ausgabe des LIFO-Tests bei Ausführung:

There was 1 failure:

1) popInvertsPush(StackTest)

```
java.lang.AssertionError: expected:<749398> but was:<null>  
    at org.junit.Assert.fail(Assert.java:71) ...  
    at org.junit.Assert.assertEquals(Assert.java:116)  
    at StackTest.popInvertsPush(StackTest.java:26)
```

9. popInvertsPush-Eigenschaft herstellen

```
private final String[] entries = new String[10];  
private int fill = 0;
```

```
@Override  
public void push(final String element) {  
    entries[fill] = element;  
    fill++;  
}
```

```
@Override  
public String pop() throws EmptyExc {  
    fill--;  
    return entries[fill];  
}
```

¿? So korrekte Implementierung?

10. Testtreiber ergänzen: Nach 4 push()s und 4 pop()s muss Stack wieder leer sein

```
@Test public void emptyAfterPushsAndPops()  
throws Exception{  
    popInvertsPush();  
    try{  
        stack.pop();  
        fail("Stack.EmptyExc expected");  
    } catch ( Stack.EmptyExc expected ){  
    }  
}
```

Test-Ausgabe von JUnit:

There was 1 failure:

1) emptyAfterPushsAndPops(StackTest)

java.lang.ArrayIndexOutOfBoundsException: -1

at Stack.pop(Stack.java:16)

at StackTest.emptyAfterPushsAndPops(StackTest.java:33)

11. emptyAfterPushsAndPops befriedigen:

Vor dem fehlgeschlagenen Array-Zugriff bauen wir eine Wächterklausel¹ ein.

```
@Override
public String pop() throws EmptyExc {
    fill--;
    if(fill<0){ //Wächterklausel
        throw new EmptyExc();
    }
    return entries[fill];
}
```

¿? So korrekte Implementierung?

¹ <https://wiki.c2.com/?GuardClause>

12. Testtreiber ergänzen: pop() auf leerem Stack läßt ihn leer

```
@Test public void popOnEmptyDoesNotModify()  
throws Stack.EmptyExc {  
    try{stack.pop(); fail("Stack.EmptyExc expected");}  
    catch(Stack.EmptyExc expected){}  
    popInvertsPush();  
    try{stack.pop(); fail("Stack.EmptyExc expected");}  
    catch(Stack.EmptyExc expected){}  
}
```

Test-Ausgabe von JUnit:

There was 1 failure:

1) popOnEmptyDoesNotModify(StackTest)

java.lang.ArrayIndexOutOfBoundsException: -1

at Stack.push(Stack.java:7)

at StackTest.popInvertsPush(StackTest.java:22)

at StackTest.popOnEmptyDoesNotModify(StackTest.java:41)

13. popOnEmptyDoesNotModify befriedigen:

@Override

```
public String pop() throws EmptyExc {  
    if(fill <= 0){ //Wächterklausel vorgezogen  
        throw new EmptyExc();  
    }  
    fill--;  
    return entries[fill];  
}
```

Jetzt verhält sich der Stack korrekt. Kein weiterer Änderungsbedarf.

14. Round Trip Test

- Eine reine Funktion (ohne Seiteneffekt) kann man für sich testen.
- Methoden eines Objektes mit Gedächtnis nur im Zusammenspiel testbar: Bei `popInvertsPush()` wird Stack erst mit `push` gefüllt, dann mit `pop` ausgelesen.
- Bei Datenbanktests sollte Hineinschreiben und Auslesen in verschiedenen Transaktionen erfolgen!

15. Best Practice Quellcodeorganisation

Testsuite in eigenem Verzeichnisbaum unter `src/test` parallel zum Produktverzeichnisbaum unter `src/main`.

Maven führt bei `mvn test` alle Klassen, deren Name auf `Test` endet, als Testtreiber aus.

Eclipse führt mittels *MausRechts* > *Run As* > *JUnit Test* alle Testtreiber im Verzeichnis aus.

- ☞ Projekt Aufgabenplaner Testsuite ausführen
 - in Maven
 - in IDE.

❓ Lernen: Wie Regular Expression für E-Mail-Adr.?

16. Testtreiber als Lernmittel

Bsp.: Lernen von Regular Expressions

Schreiben mehrerer Testfälle:

Zeichenkette als eMail-Adresse	Korrekt?
Franz	Nein
@	Nein
Franz@	Nein
Franz@hallo	Nein (?)
Franz@hallo.de	Ja
Franz@hAlLo.De	Ja
franz@hallo. de	Nein

Email-Muster-Testfälle in JUnit

```
public class EMailTest extends Assert {
```

```
    private static final Pattern emailPattern = Pattern.compile("?");
```

```
    @Test public void eMailAdressen(){  
        assertEmail("Franz@hallo", true);  
        assertEmail("Franz@hallo.de", true);  
        assertEmail("Franz@hAlLo.De", true);  
        assertEmail("Franz", false);  
        assertEmail("Franz@", false);  
        assertEmail("hallo.de", false);  
        assertEmail("franz@hallo .de", false);  
    }
```

```
    private void assertEmail(final String toCheck, boolean isEmail){  
        final Matcher m = emailPattern.matcher(toCheck);  
        assertEquals(isEmail, m.matches());  
    }
```

17. Test-Suite zum Ausführen von Testtreibern in definierter Reihenfolge

```
package fb6._any.ut;
import org.junit.runner.RunWith;
import org.junit.runners.Suite;

/**Testsuite zum zusammengefassten Ausführen bestimmter
Tests.*/
@RunWith (Suite.class)
@Suite.SuiteClasses ({
    FileUtilTest.class,
    StringUtilTest.class
})
public class TestsAnyUt {}
```

18. Fazit

- Testfälle erhöhen die Programmqualität.
- Reproduzierbare Testfälle erleichtern Änderungen am Code.
- „Test First“ (erst Testtreiber schreiben, dann befriedigen) erhöht die Testabdeckung und damit die Qualität und Änderungsfreundlichkeit.
- JUnit ist das bewährteste Framework zur Testautomation, vielfach kopiert.