Test-First

Mark Wiesemann

14. September 2005

Übersicht

Übersicht

Beispiel: Money-Klasse

1. Testfall

```
import junit.framework.TestCase;

public class MoneyTest extends TestCase {

   public void testAdd10() {
      Money money = new Money();
      money.add(10);
      assertEquals(10, money.get());
   }

}
```

Beispiel: Money-Klasse

Code zum 1. Testfall

```
public class Money {

  public void add(int amount) {
  }

  public int get() {
    return 10;
  }

}
```

Beispiel: Money-Klasse

2. Testfall

```
import junit.framework.TestCase;

public class MoneyTest extends TestCase {

  public void testAdd10() {
    Money money = new Money();
    money.add(10);
    assertEquals(10, money.get());
  }

  public void testAdd20() {
    Money money = new Money();
    money.add(20);
    assertEquals(20, money.get());
  }

}
```

Beispiel: Money-Klasse

Code zum 2. Testfall

```
public class Money {

  private int totalAmount;

  public void add(int amount) {
    totalAmount = amount;
  }

  public int get() {
    return totalAmount;
  }

}
```

Beispiel: Money-Klasse

3. Testfall

```
import junit.framework.TestCase;

public class MoneyTest extends TestCase {

   ...

   public void testAdd10And20() {
      Money money = new Money();
      money.add(10);
      money.add(20);
      assertEquals(30, money.get());
   }

}
```

Beispiel: Money-Klasse

Code zum 3. Testfall

```
public class Money {

  private int totalAmount;

  public void add(int amount) {
    totalAmount = totalAmount + amount;
  }

  public int get() {
    return totalAmount;
  }

}
```

Beispiel: Money-Klasse

Refaktorisierter Code zum 3. Testfall

```
public class Money {

  private int totalAmount;

  public void add(int amount) {
    totalAmount += amount;
  }

  public int get() {
    return totalAmount;
  }

}
```

Beispiel: Money-Klasse

4. Testfall

```
import junit.framework.TestCase;

public class MoneyTest extends TestCase {

   ...

   public void testRemoveWithException() {
      Money money = new Money();
      money.add(10);
      money.add(20);
      try {
         money.remove(50);
         fail();
      } catch(Exception e) {}
   }

}
```

Beispiel: Money-Klasse

Code zum 4. Testfall

```
public class Money {

  private int totalAmount;

  public void add(int amount) {
    totalAmount += amount;
  }

  public void remove(int amount) throws Exception {
    throw new Exception("Not␣enough␣money");
  }

  public int get() {
    return totalAmount;
  }

}
```

Vorgehen

- zuerst Test schreiben, dann den eigentlichen Code
- nur so viel Code schreiben, wie der Test verlangt
- Entwicklung in kleinen Schritten (10-Minuten-Iterationen)
- nach jeder Iteration: Refaktorisieren, Duplikate entfernen
- vor der Integration müssen alle Tests erfolgreich laufen

Vorgehen

- zuerst Test schreiben, dann den eigentlichen Code
- nur so viel Code schreiben, wie der Test verlangt
- Entwicklung in kleinen Schritten (10-Minuten-Iterationen)
- nach jeder Iteration: Refaktorisieren, Duplikate entfernen
- vor der Integration müssen alle Tests erfolgreich laufen

Vorgehen

- zuerst Test schreiben, dann den eigentlichen Code
- nur so viel Code schreiben, wie der Test verlangt
- Entwicklung in kleinen Schritten (10-Minuten-Iterationen)
- nach jeder Iteration: Refaktorisieren, Duplikate entfernen
- vor der Integration müssen alle Tests erfolgreich laufen

Vorgehen

- zuerst Test schreiben, dann den eigentlichen Code
- nur so viel Code schreiben, wie der Test verlangt
- Entwicklung in kleinen Schritten (10-Minuten-Iterationen)
- nach jeder Iteration: Refaktorisieren, Duplikate entfernen
- vor der Integration müssen alle Tests erfolgreich laufen

Vorgehen

- zuerst Test schreiben, dann den eigentlichen Code
- nur so viel Code schreiben, wie der Test verlangt
- Entwicklung in kleinen Schritten (10-Minuten-Iterationen)
- nach jeder Iteration: Refaktorisieren, Duplikate entfernen
- vor der Integration müssen alle Tests erfolgreich laufen

Regeln

Minimum-Regel

Wenn der Test läuft, steht der Code.

Farb-Regel

- vom grünen Testbalken zum roten Testbalken hin arbeiten
- auf dem kürzesten Weg zurück zum grünen Testbalken arbeiten
- am grünen Testbalken angekommen: Refaktorisieren

Regeln

Minimum-Regel

Wenn der Test läuft, steht der Code.

Farb-Regel

- vom grünen Testbalken zum roten Testbalken hin arbeiten
- auf dem kürzesten Weg zurück zum grünen Testbalken arbeiten
- am grünen Testbalken angekommen: Refaktorisieren

Vorteile

- Tests dokumentieren Code
- schnelles Feedback $\Rightarrow$ weniger Fehler
- besseres und einfacheres Design
- weitreichende Änderungen werden erleichtert
- Vermeidung von „Featuritis“
- jedes Stück Code ist getestet

Vorteile

- Tests dokumentieren Code
- schnelles Feedback $\Rightarrow$ weniger Fehler
- besseres und einfacheres Design
- weitreichende Änderungen werden erleichtert
- Vermeidung von „Featuritis“
- jedes Stück Code ist getestet

Vorteile

- Tests dokumentieren Code
- schnelles Feedback $\Rightarrow$ weniger Fehler
- besseres und einfacheres Design
- weitreichende Änderungen werden erleichtert
- Vermeidung von „Featuritis“
- jedes Stück Code ist getestet

Vorteile

- Tests dokumentieren Code
- schnelles Feedback $\Rightarrow$ weniger Fehler
- besseres und einfacheres Design
- weitreichende Änderungen werden erleichtert
- Vermeidung von „Featuritis“
- jedes Stück Code ist getestet

Vorteile

- Tests dokumentieren Code
- schnelles Feedback $\Rightarrow$ weniger Fehler
- besseres und einfacheres Design
- weitreichende Änderungen werden erleichtert
- Vermeidung von „Featuritis“
- jedes Stück Code ist getestet

Vorteile

- Tests dokumentieren Code
- schnelles Feedback $\Rightarrow$ weniger Fehler
- besseres und einfacheres Design
- weitreichende Änderungen werden erleichtert
- Vermeidung von „Featuritis“
- jedes Stück Code ist getestet

Übersicht

Schwarze und weiße Kisten

Black-Box-Tests

- spezifikationsbasierte Tests
- Erstellung vor der Implementierung möglich

White-Box-Tests

- implementierungsbasierte Tests
- Erstellung erst nach der Implementierung möglich

Schwarze und weiße Kisten

Black-Box-Tests

- spezifikationsbasierte Tests
- Erstellung vor der Implementierung möglich

White-Box-Tests

- implementierungsbasierte Tests
- Erstellung erst nach der Implementierung möglich

Der falsche Weg

Falsche Regel

Schreibe für jede öffentliche Methode einen Testfall.

- Fehler durch Zustandsänderungen von Objekten werden nicht entdeckt
- meistens reicht ein Testfall nicht

Richtige Regel

Teste jede typische Verwendung eines Objekts.

Der falsche Weg

Falsche Regel

Schreibe für jede öffentliche Methode einen Testfall.

- Fehler durch Zustandsänderungen von Objekten werden nicht entdeckt
- meistens reicht ein Testfall nicht

Richtige Regel

Teste jede typische Verwendung eines Objekts.

Der falsche Weg

Falsche Regel

Schreibe für jede öffentliche Methode einen Testfall.

- Fehler durch Zustandsänderungen von Objekten werden nicht entdeckt
- meistens reicht ein Testfall nicht

Richtige Regel

Teste jede typische Verwendung eines Objekts.

Der falsche Weg

Falsche Regel

Schreibe für jede öffentliche Methode einen Testfall.

- Fehler durch Zustandsänderungen von Objekten werden nicht entdeckt
- meistens reicht ein Testfall nicht

Richtige Regel

Teste jede typische Verwendung eines Objekts.

Testideen

Ein Test pro Idee

- schnellere Erstellung
- besser lesbar
- leichter umbaubar
- nur das, was beabsichtigt ist, wird getestet

Viele Ideen pro Test

- mehr Planung erforderlich
- fehleranfälliger
- schwieriger umbaubar
- mehr als das, was beabsichtigt ist, wird getestet
  $\Rightarrow$ „Pures Glück“

Daher:

- zunächst viele Testfälle schreiben
- bei Bedarf komplexere Testfälle ergänzen

Testideen

Ein Test pro Idee

- schnellere Erstellung
- besser lesbar
- leichter umbaubar
- nur das, was beabsichtigt ist, wird getestet

Viele Ideen pro Test

- mehr Planung erforderlich
- fehleranfälliger
- schwieriger umbaubar
- mehr als das, was beabsichtigt ist, wird getestet
  $\Rightarrow$ „Pures Glück“

Daher:

- zunächst viele Testfälle schreiben

Testideen

Ein Test pro Idee

- schnellere Erstellung
- besser lesbar
- leichter umbaubar
- nur das, was beabsichtigt ist, wird getestet

Viele Ideen pro Test

- mehr Planung erforderlich
- fehleranfälliger
- schwieriger umbaubar
- mehr als das, was beabsichtigt ist, wird getestet
  $\Rightarrow$ „Pures Glück“

Daher:

- zunächst kleine Testfälle schreiben
- ggf. danach komplexere Testfälle ergänzen

Testideen

Ein Test pro Idee

- schnellere Erstellung
- besser lesbar
- leichter umbaubar
- nur das, was beabsichtigt ist, wird getestet

Viele Ideen pro Test

- mehr Planung erforderlich
- fehleranfälliger
- schwieriger umbaubar
- mehr als das, was beabsichtigt ist, wird getestet
  $\Rightarrow$ „Pures Glück“

Daher:

- zunächst kleine Testfälle schreiben
- danach komplexere Testfälle ergänzen

Testideen

Ein Test pro Idee

- schnellere Erstellung
- besser lesbar
- leichter umbaubar
- nur das, was beabsichtigt ist, wird getestet

Viele Ideen pro Test

- mehr Planung erforderlich
- fehleranfälliger
- schwieriger umbaubar
- mehr als das, was beabsichtigt ist, wird getestet
  $\Rightarrow$ „Pures Glück“

Daher:

- zunächst kleine Testfälle schreiben
- danach komplexere Testfälle ergänzen

Testideen

Ein Test pro Idee

- schnellere Erstellung
- besser lesbar
- leichter umbaubar
- nur das, was beabsichtigt ist, wird getestet

Viele Ideen pro Test

- mehr Planung erforderlich
- fehleranfälliger
- schwieriger umbaubar
- mehr als das, was beabsichtigt ist, wird getestet
  $\Rightarrow$ „Pures Glück“

Daher:

- zunächst kleine Testfälle schreiben
- danach komplexere Testfälle ergänzen

Testfälle finden

- Vor- und Nachbedingungen von Methoden testen
- jede Exception mindestens einmal auslösen
- bei Arrays: `null` und leere Arrays testen
- auch mit der abstraktesten Klasse testen, die laut Methodensignatur zulässig ist (z.B. `Object`)

Error-Guessing

- Erfahrung zeigt:
  - manche Mitarbeiter finden mehr Fehler als andere
- kein systematisches Verfahren
- „intuitives Testen“

Testfälle finden

- Vor- und Nachbedingungen von Methoden testen
- jede Exception mindestens einmal auslösen
- bei Arrays: `null` und leere Arrays testen
- auch mit der abstraktesten Klasse testen, die laut Methodensignatur zulässig ist (z.B. `Object`)

Error-Guessing

- Erfahrung zeigt:
  - manche Mitarbeiter finden mehr Fehler als andere
- kein systematisches Verfahren
- „intuitives Testen“

Testfälle finden

- Vor- und Nachbedingungen von Methoden testen
- jede Exception mindestens einmal auslösen
- bei Arrays: `null` und leere Arrays testen
- auch mit der abstraktesten Klasse testen, die laut Methodensignatur zulässig ist (z.B. `Object`)

Error-Guessing

- Erfahrung zeigt:
  - manche Mitarbeiter finden mehr Fehler als andere
- kein systematisches Verfahren
- „intuitives Testen“

Testfälle finden

- Vor- und Nachbedingungen von Methoden testen
- jede Exception mindestens einmal auslösen
- bei Arrays: `null` und leere Arrays testen
- auch mit der abstraktesten Klasse testen, die laut Methodensignatur zulässig ist (z.B. `Object`)

Error-Guessing

- Erfahrung zeigt:
  - manche Mitarbeiter finden mehr Fehler als andere
- kein systematisches Verfahren
- „intuitives Testen“

Testfälle finden

- Vor- und Nachbedingungen von Methoden testen
- jede Exception mindestens einmal auslösen
- bei Arrays: `null` und leere Arrays testen
- auch mit der abstraktesten Klasse testen, die laut Methodensignatur zulässig ist (z.B. `Object`)

Error-Guessing

- Erfahrung zeigt:
  - manche Mitarbeiter finden mehr Fehler als andere
- kein systematisches Verfahren
- „intuitives Testen“

Grenzwerte

- Ränder von Wertebereichen müssen verstärkt getestet werden
- 0, 1, 2 und `MAXINTEGER` sind besser geeignet als 5, 12, 69 und 101
- leere und sehr lange Strings sind besser geeignet als Strings der Längen 10 und 100
- Größe von Eingabedateien: obere (z.B. 500 MB) und untere Grenzen (0 Byte) testen

Grenzwerte

- Ränder von Wertebereichen müssen verstärkt getestet werden
- 0, 1, 2 und `MAXINTEGER` sind besser geeignet als 5, 12, 69 und 101
- leere und sehr lange Strings sind besser geeignet als Strings der Längen 10 und 100
- Größe von Eingabedateien: obere (z.B. 500 MB) und untere Grenzen (0 Byte) testen

Grenzwerte

- Ränder von Wertebereichen müssen verstärkt getestet werden
- 0, 1, 2 und `MAXINTEGER` sind besser geeignet als 5, 12, 69 und 101
- leere und sehr lange Strings sind besser geeignet als Strings der Längen 10 und 100
- Größe von Eingabedateien: obere (z.B. 500 MB) und untere Grenzen (0 Byte) testen

Grenzwerte

- Ränder von Wertebereichen müssen verstärkt getestet werden
- 0, 1, 2 und `MAXINTEGER` sind besser geeignet als 5, 12, 69 und 101
- leere und sehr lange Strings sind besser geeignet als Strings der Längen 10 und 100
- Größe von Eingabedateien: obere (z.B. 500 MB) und untere Grenzen (0 Byte) testen

Äquivalenzklassen 1

Sortierfunktion

```
private final static int MIN_QUICKSORT = 15;
public List sort(List unsorted) {
  if (unsorted.size() < MIN_QUICKSORT) {
    return bubbleSort(unsorted);
  } else {
    return quickSort(unsorted);
  }
}
```

- nicht nur Grenzfälle 0, 1 und `MAXINTEGER` testen
- auch Grenzfälle 14 und 15 testen

Äquivalenzklassen 1

Sortierfunktion

```
private final static int MIN_QUICKSORT = 15;
public List sort(List unsorted) {
  if (unsorted.size() < MIN_QUICKSORT) {
    return bubbleSort(unsorted);
  } else {
    return quickSort(unsorted);
  }
}
```

- nicht nur Grenzfälle 0, 1 und `MAXINTEGER` testen
- auch Grenzfälle 14 und 15 testen

Äquivalenzklassen 1

Sortierfunktion

```
private final static int MIN_QUICKSORT = 15;
public List sort(List unsorted) {
  if (unsorted.size() < MIN_QUICKSORT) {
    return bubbleSort(unsorted);
  } else {
    return quickSort(unsorted);
  }
}
```

- nicht nur Grenzfälle 0, 1 und `MAXINTEGER` testen
- auch Grenzfälle 14 und 15 testen

Äquivalenzklassen 2

Definition

Eine Gruppe von Tests formt eine Äquivalenzklasse:

- wenn alle das Gleiche testen,
- wenn ein Test einen Fehler findet, die anderen das (wahrscheinlich) auch tun
- wenn ein Test keinen Fehler findet, die anderen das (wahrscheinlich) auch nicht tun

Äquivalenzklassen 2

Definition

Eine Gruppe von Tests formt eine Äquivalenzklasse:

- wenn alle das Gleiche testen,
- wenn ein Test einen Fehler findet, die anderen das (wahrscheinlich) auch tun
- wenn ein Test keinen Fehler findet, die anderen das (wahrscheinlich) auch nicht tun

Äquivalenzklassen 2

Definition

Eine Gruppe von Tests formt eine Äquivalenzklasse:

- wenn alle das Gleiche testen,
- wenn ein Test einen Fehler findet, die anderen das (wahrscheinlich) auch tun
- wenn ein Test keinen Fehler findet, die anderen das (wahrscheinlich) auch nicht tun

Übersicht

Warmup-Aufgabe

Code aus der Aufgabe

```
assertNotNull("The one vertice is there.",
    onePointLine.getVerticeAt(1));
```

Einfache Lösung

```
assertNotNull("The one vertice is there.",
    onePointLine.getVerticeAt(0));
```

Warmup-Aufgabe

Code aus der Aufgabe

```
assertNotNull("The one vertice is there.",
    onePointLine.getVerticeAt(1));
```

Einfache Lösung

```
assertNotNull("The one vertice is there.",
    onePointLine.getVerticeAt(0));
```

Warmup-Aufgabe

Code aus der Aufgabe

```
assertNotNull("The one vertice is there.",
    onePointLine.getVerticeAt(1));
```

Punkt 1 der Checkliste für Penible

Kannst Du den Test so ergänzen, dass sichergestellt ist, dass dieser Exception geworfen wird?

Lösung für Penible

```
try {
  onePointLine.getVerticeAt(1);
  fail();
} catch(ArrayIndexOutOfBoundsException e) {}
```

Warmup-Aufgabe

Code aus der Aufgabe

```
assertNotNull("The␣one␣vertice␣is␣there.",
    onePointLine.getVerticeAt(1));
```

Punkt 1 der Checkliste für Penible

Kannst Du den Test so ergänzen, dass sichergestellt ist, dass dieser Exception geworfen wird?

Lösung für Penible

```
try {
  onePointLine.getVerticeAt(1);
  fail();
} catch(ArrayIndexOutOfBoundsException e) {}
```

Grundsätzliches zu JUnit

- Reflection-API liefert
  - öffentliche
  - nicht statische
  - parameterlose

  Methoden, deren Namen mit `test` beginnen
- Methoden werden isoliert voneinander aufgerufen
- neues Exemplar der Testklasse für jeden Testfall

Grundsätzliches zu JUnit

- Reflection-API liefert
  - öffentliche
  - nicht statische
  - parameterlose

  Methoden, deren Namen mit `test` beginnen
- Methoden werden isoliert voneinander aufgerufen
- neues Exemplar der Testklasse für jeden Testfall

Grundsätzliches zu JUnit

- Reflection-API liefert
  - öffentliche
  - nicht statische
  - parameterlose

  Methoden, deren Namen mit `test` beginnen
- Methoden werden isoliert voneinander aufgerufen
- neues Exemplar der Testklasse für jeden Testfall

Lebenszyklus eines Testfalls

1. Aufruf der `setUp`-Methode
2. Aufruf der `test...`-Methode
3. Aufruf der `tearDown`-Methode

Beachte:

Ausführungsreihenfolge
!=
Reihenfolge der Testfälle im Code

Lebenszyklus eines Testfalls

1. Aufruf der `setUp`-Methode
2. Aufruf der `test...`-Methode
3. Aufruf der `tearDown`-Methode

Beachte:

Ausführungsreihenfolge
!=
Reihenfolge der Testfälle im Code

Lebenszyklus eines Testfalls

1. Aufruf der `setUp`-Methode
2. Aufruf der `test...`-Methode
3. Aufruf der `tearDown`-Methode

Beachte:

Ausführungsreihenfolge
!=
Reihenfolge der Testfälle im Code

Lebenszyklus eines Testfalls

1. Aufruf der `setUp`-Methode
2. Aufruf der `test...`-Methode
3. Aufruf der `tearDown`-Methode

Beachte:

Ausführungsreihenfolge
!=
Reihenfolge der Testfälle im Code

Methoden der Assert-Klasse
(Auswahl)

```
assertEquals(boolean expected, boolean actual)
assertEquals(double expected, double actual,
  double delta)
assertFalse(boolean condition)
assertTrue(boolean condition)
assertNull(java.lang.Object object)
assertNotNull(java.lang.Object object)
assertSame(java.lang.Object expected,
  java.lang.Object actual)
assertNotSame(java.lang.Object expected,
  java.lang.Object actual)
fail()
```

Beachte:

Erst den erwarteten, dann den aktuellen Wert angeben.

Methoden der Assert-Klasse 2
(Auswahl)

```
assertEquals(java.lang.String message,
  boolean expected, boolean actual)
```

- optionale Angabe einer Forderung bei jeder `assert`-Methode
- Empfehlung: positive Forderung im Konjunktiv formulieren

Beispiele

- `PolyLine should have been created.`
- `After adding another vertice the length should be 3.`

Methoden der Assert-Klasse 2
(Auswahl)

```
assertEquals(java.lang.String message,
  boolean expected, boolean actual)
```

- optionale Angabe einer Forderung bei jeder `assert`-Methode
- Empfehlung: positive Forderung im Konjunktiv formulieren

Beispiele

- `PolyLine should have been created.`
- `After adding another vertice the length should be 3.`

Methoden der Assert-Klasse 2
(Auswahl)

```
assertEquals(java.lang.String message,
  boolean expected, boolean actual)
```

- optionale Angabe einer Forderung bei jeder `assert`-Methode
- Empfehlung: positive Forderung im Konjunktiv formulieren

Beispiele

- `PolyLine should have been created.`
- `After adding another vertice the length should be 3.`

Methoden der Assert-Klasse 2
(Auswahl)

```
assertEquals(java.lang.String message,
  boolean expected, boolean actual)
```

- optionale Angabe einer Forderung bei jeder `assert`-Methode
- Empfehlung: positive Forderung im Konjunktiv formulieren

Beispiele

- `PolyLine should have been created.`
- `After adding another vertice the length should be 3.`

Übersicht

Dummy- und Mock-Objekte

- in den meisten Anwendungen hängen Objekte voneinander ab
- Bottom-Up-Ansatz:
  - Test und Entwicklung beginnt mit den Klassen, die nur von systemeigenen Klassen abhängen
- aber: Top-Down-Vorgehen beim Test-First-Ansatz
- daher: Verwendung von Dummy- oder Mock-Objekten, die die echte Implementierung für den Test ersetzen

Dummy- und Mock-Objekte

- in den meisten Anwendungen hängen Objekte voneinander ab
- Bottom-Up-Ansatz:
  - Test und Entwicklung beginnt mit den Klassen, die nur von systemeigenen Klassen abhängen
- aber: Top-Down-Vorgehen beim Test-First-Ansatz
- daher: Verwendung von Dummy- oder Mock-Objekten, die die echte Implementierung für den Test ersetzen

Dummy- und Mock-Objekte

- in den meisten Anwendungen hängen Objekte voneinander ab
- Bottom-Up-Ansatz:
  - Test und Entwicklung beginnt mit den Klassen, die nur von systemeigenen Klassen abhängen
- aber: Top-Down-Vorgehen beim Test-First-Ansatz
- daher: Verwendung von Dummy- oder Mock-Objekten, die die echte Implementierung für den Test ersetzen

Dummy- und Mock-Objekte

- in den meisten Anwendungen hängen Objekte voneinander ab
- Bottom-Up-Ansatz:
  - Test und Entwicklung beginnt mit den Klassen, die nur von systemeigenen Klassen abhängen
- aber: Top-Down-Vorgehen beim Test-First-Ansatz
- daher: Verwendung von Dummy- oder Mock-Objekten, die die echte Implementierung für den Test ersetzen

Cultivate-Plugin

- entstanden in den letzten beiden XP-Praktika
- Eclipse-Plugin zur Verbesserung von Java-Software
- berechnet eine einfache Metrik zur statischen Testabdeckung
- „Wie viel Prozent der öffentlichen Methoden werden getestet?“

Cultivate-Plugin

- entstanden in den letzten beiden XP-Praktika
- Eclipse-Plugin zur Verbesserung von Java-Software
- berechnet eine einfache Metrik zur statischen Testabdeckung
- „Wie viel Prozent der öffentlichen Methoden werden getestet?“

Cultivate-Plugin

- entstanden in den letzten beiden XP-Praktika
- Eclipse-Plugin zur Verbesserung von Java-Software
- berechnet eine einfache Metrik zur statischen Testabdeckung
- „Wie viel Prozent der öffentlichen Methoden werden getestet?“

Cultivate-Plugin

- entstanden in den letzten beiden XP-Praktika
- Eclipse-Plugin zur Verbesserung von Java-Software
- berechnet eine einfache Metrik zur statischen Testabdeckung
- „Wie viel Prozent der öffentlichen Methoden werden getestet?“

Fragen?