Über den Unterschied, den drei Refactorings ausmachen können

Während einer Vorlesung über Unit Tests entwickelten die Studenten mit mir im Rahmen einer praktischen Übung einige Tests mit JUnit 4 für eine Klasse „Konto“. Die zu testende Klasse war eher unspannend – für einen gegebenen Kunden wurde ein Betrag verwaltet. Am Konto konnten Abhebungen durchgeführt werden, allerdings konnten nur manche Kunden ihr Konto dabei überziehen.

Der erste Test für das Abheben arbeitet mit einem Mock-Objekt als Kunden. Für das Mock-Objekt verwendeten wir das relativ einfach zu verstehende Framework „EasyMock“:

```
@Test
public void kannAbheben() {
	Kunde kunde = EasyMock.createMock(Kunde.class);
	EasyMock.expect(kunde.kannUeberziehen()).andReturn(true);
	EasyMock.replay(kunde);
	Konto konto = new Konto(kunde);
	Euro required = new Euro(30);

	Euro bargeld = konto.hebeAb(required);

	assertEquals(new Euro(30), bargeld);
	assertEquals(new Euro(-30), konto.stand());
	EasyMock.verify(kunde);
}
```

Der zweite Test stellt das Verhalten von Konto sicher, falls der Kunde sein Konto nicht überziehen darf. In diesem Fall entschieden wir uns dazu, dem Kunden 0 Euro Bargeld auszuzahlen:

```
@Test
public void kannNichtAbheben() {
	Kunde kunde = EasyMock.createMock(Kunde.class);
	EasyMock.expect(kunde.kannUeberziehen()).andReturn(false);
	EasyMock.replay(kunde);
	Konto konto = new Konto(kunde);
	Euro required = new Euro(30);

	Euro bargeld = konto.hebeAb(required);

	assertEquals(Euro.ZERO, bargeld);
	assertEquals(Euro.ZERO, konto.stand());
	EasyMock.verify(kunde);
}
```

Natürlich war dieser zweite Test durch stupides Kopieren & Einfügen zustande gekommen, worauf auch die in diesem Kontext unglückliche Benennung der lokalen Variable „required“ hindeutet. Ziel der Übung war bis zu diesem Punkt allerdings, die Verwendung von EasyMock zu erklären, nicht die Tests perfekt zu verfassen.

Nachdem wir die fünf Stufen eines normalisierten mock-basierten Unit Tests besprochen hatten (Initialisieren, Mocks einlernen, Testcode ausführen, Ergebnisse überprüfen, Mocks verifizieren), formulierte ich meinen Unmut über die offensichtliche Duplikation und die schlechte Code-Qualität. Ich überließ es den Studenten, uns aus dieser Situation herauszuholen. Ein Student, namentlich Nils Schwenkel*, begann, Vorschläge zu machen:

*) Namensnennung mit Genehmigung

Erstes Refactoring: Extract Method

Warnung: Ich verwende für Code-Beispiele in den Vorlesungen immer einen kruden Mix aus deutschen Klassen- und Methodennamen und englischen Begriffen. Die jeweilige Wahl der Sprache ist mir manchmal nicht bewusst und führt zu seltsamen Namen wie beispielsweise „performAbhebenTestMit“. Ich bitte, diese Verwirrung gnädig zu übersehen.

Der erste Schritt, den der Student gehen wollte, war das Auslagern der Gemeinsamkeiten der beiden Test-Methoden in einer neuen Methode. Die Unterschiede zwischen den beiden Tests werden durch Methoden-Parameter behandelt. Die entstandene Methode las sich wie folgt:

```
protected void performAbhebenTestMit(
            boolean kundeKannUeberziehen,
            Euro abhebeBetrag,
            Euro erwarteterBetrag,
            Euro erwarteterKontostand) {
      Kunde kunde = EasyMock.createMock(Kunde.class);
      EasyMock.expect(kunde.kannUeberziehen()).andReturn(kundeKannUeberziehen);
      EasyMock.replay(kunde);
      Konto konto = new Konto(kunde);

      Euro bargeld = konto.hebeAb(abhebeBetrag);

      assertEquals(erwarteterBetrag, bargeld);
      assertEquals(erwarteterKontostand, konto.stand());
      EasyMock.verify(kunde);
}
```

Damit verkürzten sich die zwei Test-Methoden erheblich:

```
@Test
public void kannAbheben() {
      performAbhebenTestMit(
                  true,
                  new Euro(30),
                  new Euro(30),
                  new Euro(-30));
}
```

```
@Test
public void kannNichtAbheben() {
      performAbhebenTestMit(
                  false,
                  new Euro(30),
                  Euro.ZERO,
                  Euro.ZERO);
}
```

Damit war die Duplikation aufgelöst, die Test-Methoden allerdings so zusammengeschrumpft, dass sie nicht mehr verständlich waren. Darüber waren wir nicht besonders glücklich, sahen es aber nur als Zwischenschritt. Denn es folgte das zweite Refactoring.

Zweites Refactoring: Introduce Explaining Variable

Der zweite Schritt bestand darin, den namenlosen Parametern in den beiden Methoden-Aufrufen durch Einfügen von lokalen Variablen eine klarere Bedeutung zu geben:

```
@Test
public void kannAbheben() {
      boolean kannAbheben = true;
      Euro abhebeBetrag = new Euro(30);
      Euro auszahlung = new Euro(30);
      Euro kontostand = new Euro(-30);
      performAbhebenTestMit(
                  kannAbheben,
                  abhebeBetrag,
                  auszahlung,
                  kontostand);
}
```

```
@Test
public void kannNichtAbheben() {
      boolean kannAbheben = false;
      Euro abhebeBetrag = new Euro(30);
      Euro auszahlung = Euro.ZERO;
      Euro kontostand = Euro.ZERO;
      performAbhebenTestMit(
                  kannAbheben,
                  abhebeBetrag,
                  auszahlung,
                  kontostand);
}
```

Damit war die Bedeutung der einzelnen Parameter geklärt, die Test-Methoden allerdings immer noch nicht wirklich lesbarer. Der Code war jetzt verständlich, im Sinne von „nicht kryptisch“, aber nicht intuitiv erfassbar.
Meiner Meinung nach ist Code dann wirklich lesbar, wenn er selbst von Laien direkt begriffen werden kann, wenn sich also die Arbeit für die Entzifferung auf dem Niveau von „Durchlesen“ bewegt.

Ich präsentierte dem Studenten dieses Ziel, woraufhin er kurz zögerte, dann aber das dritte und entscheidende Refactoring vorschlug:

Drittes Refactoring: Rename (Variable)

Der dritte Schritt verbessert nur noch die Lesbarkeit des Codes. Die anfängliche Duplikation ist aufgelöst und der unverständliche Teil der Test-Methoden ist bereits mit Erklärungen versehen. Dieser Schritt ist also optional, macht aber einen erheblichen Unterschied. Wir benennen die erklärenden Variablen nicht mehr nach ihrer Funktion in der aufgerufenen Methode, sondern nach ihrer Bedeutung für den Test:

```
@Test
public void kannAbheben() {
      boolean einemKundenDerÜberziehenKann = true;
      Euro erHebt30EuroAb = new Euro(30);
      Euro erhältDenVollenBetrag = new Euro(30);
      Euro undIstJetzt30EuroInDenMiesen = new Euro(-30);

      performAbhebenTestMit(
                  einemKundenDerÜberziehenKann,
                  erHebt30EuroAb,
                  erhältDenVollenBetrag,
                  undIstJetzt30EuroInDenMiesen);
}
```

```
@Test
public void kannNichtAbheben() {
      boolean einemKundenDerNichtÜberziehenKann = false;
      Euro erWill30EuroAbheben = new Euro(30);
      Euro erhältNichtsZurück = Euro.ZERO;
      Euro undSeinKontostandÄndertSichNicht = Euro.ZERO;

      performAbhebenTestMit(
                  einemKundenDerNichtÜberziehenKann,
                  erWill30EuroAbheben,
                  erhältNichtsZurück,
                  undSeinKontostandÄndertSichNicht);
}
```

Mit diesem letzten, einfachsten Refactoring, wird aus Code eine Geschichte, sofern man sich daran gewöhnt, einige Sonderzeichen und seltsame Formatierungen zu ignorieren. Die ersten vier Zeilen jeder Test-Methoden sind weiterhin für Laien unverständlich, aber für das grobe Verständnis der Vorgänge auch nicht notwendig. Wenn ein Leser den ihm unleserlich erscheinenden Teil wohlwollend überspringt, ist er immer noch in der Lage, den eigentlichen Test zu verstehen.

Ich danke diesem Studenten für seine Vorschläge, die ich in dieser Form nicht vorausgesehen hatte und deren Ergebnis mich außerordentlich überraschte. Ich halte meine Vorlesungen auch, weil ich der Meinung bin, oft mehr zu lernen als meine Studenten. In diesem Moment war es ganz sicher so.

24.03.2012
Daniel Lindner
Softwareschneiderei GmbH
Karlsruhe