Was ist ein Unit Test

„Unit Tests sind ein Verfahren, um das Verhalten einer Komponente isoliert, d.h. ohne ihre Abhängigkeiten zu anderen Komponenten, zu überprüfen."

Reden wir über diese Definition von Unit Tests. Sie besagt, wir wollen ein bestimmtes Verhalten in einem klar definierten Kontext überprüfen und dabei jede Form von Seiteneffekten vermeiden. Deshalb klammern wir die Abhängigkeiten zu anderen Komponenten so explizit aus. Jede weitere Komponente, von der unser zu testendes Verhalten abhängig ist, bedeutet einen weiteren Faktor, den wir kontrollieren müssen bzw. der das Ergebnis unseres Tests beeinflussen kann.

Sehen wir uns dazu ein Beispiel in Java an. In einem Baseballszenario wirft ein Pitcher einen Ball, den ein Spieler zu treffen versucht. Wir möchten einen Unit Test für die Methode hitBaseball() schreiben, bei der der Spieler einen durchschnittlich geworfenen Ball erfolgreich treffen soll. Spontan sähe unser Test so aus:

```
@Test
Public void testPlayerHitBallSuccessfully() {
        BaseballPlayer player = new BaseballPlayer(„Mike");
        Baseball baseball = new Pitcher(„Tom").throw(„average");
        assertTrue(„Ball verfehlt", player.hitBaseball(baseball));
}
```

Der Pitcher erzeugt beim Werfen einen Ball, der alle Attribute eines Balles hat, der durchschnittlich geworfen wurden. Der Spieler versucht den Ball zu treffen und wir gehen eigentlich davon aus, dass er das müsste. Das ist unsere Erwartungshaltung.
Ist das nun ein Unit Test gemäß unserer Definition oben? Ganz klar nein! Neben der Methode hitBaseball, dem Verhalten, dass wir eigentlich testen wollen, testen wir hier auch direkt noch throw(x) von Pitcher mit und die interne Implementierung von Baseball. Wenn dieser Test fehlschlägt, dann ist nicht deutlich ersichtlich, woran das liegen könnte, die Anzahl der Codestellen, die sich unerwartet verhalten ist einfach zu groß.

Probieren wir es mal anders:

```
@Test
Public void testPlayerHitBallSuccessfully() {
        BaseballPlayer player = new BaseballPlayer(„Mike");
        Baseball baseball = Mockito.mock(Baseball.class);
        Baseball.when(baseball.getSpeed()).thenReturn(90);
        assertTrue(„Ball verfehlt", player.hitBaseball(baseball));
}
```

Was ist jetzt anders? Um die Methode hitBaseball testen zu können, benötigen wir einen Baseball. Nur lassen wir uns den diesmal nicht von einem Pitcher erzeugen. Stattdessen erzeugen wir uns mit Hilfe eines Mockframeworks, hier Mockito, einen Mock, der so aussieht, wie ein Baseball.

```
        Baseball baseball = Mockito.mock(Baseball.class);
```

Wir brauchen jetzt keinen Pitcher mehr und sind damit in unserem Test auch nicht mehr davon abhängig, dass der Pitcher sich wirklich wie erwartet verhält. Durch den Mock haben wir uns selber etwas erzeugt, dass wie ein Baseball aussieht. Allerdings müssen wir nun noch sicherstellen, dass sich dieser Mock auch wie ein durchschnittlich geworfener Baseball verhält.
In diesem Beispiel wissen wir aus der Implementierung vom BaseballPlayer, dass wir den Ball nach seiner Geschwindigkeit fragen werden. Also erklären wir dem Mock, dass er 90 (mph) zu antworten hat, wenn er nach seiner Geschwindigkeit gefragt wird.

```
        Baseball.when(baseball.getSpeed()).thenReturn(90);
```

Und schon testen wir exakt nur noch das Verhalten von hitBaseball(x). Wenn wir Tests auf diese Art und Weise schreiben, dann bekommen wir als Ergebnis eine ganze Menge an Tests, die jeweils nur ein ganz bestimmtes Verhalten einer Komponente testen. Wir sagen manchmal auch:

ein Unit Test testet nur einen einzelnen Aspekt einer Komponente