4. Unit Testing

Was sind Unit Tests?

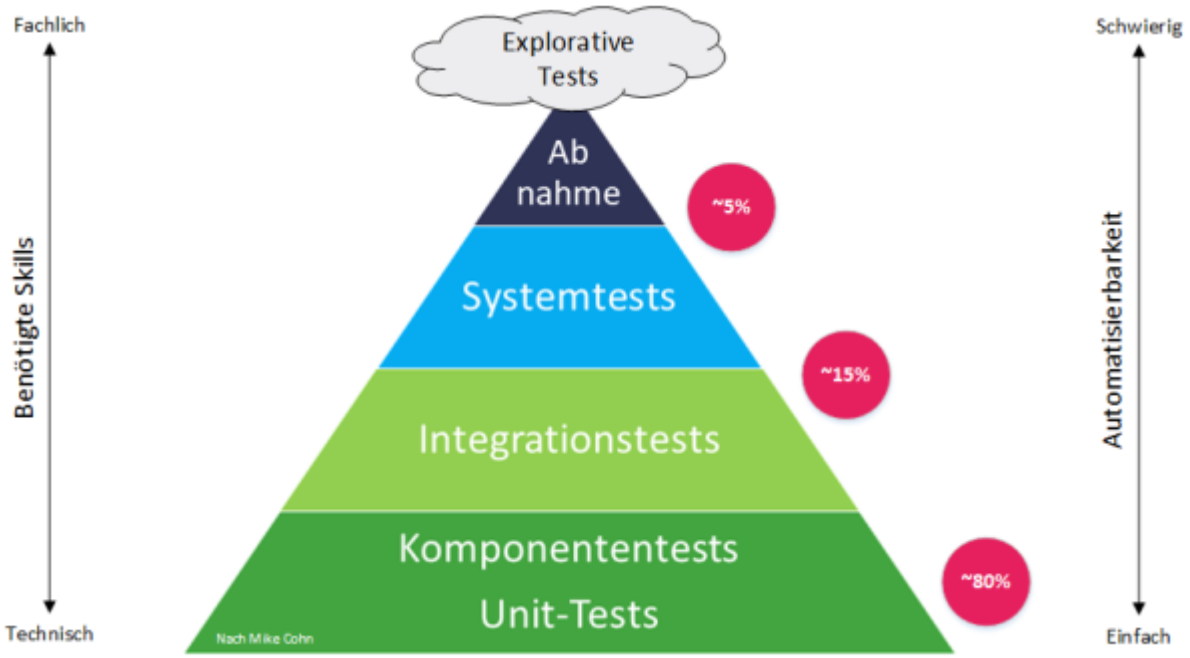
Unit Tests resp. **Komponententests** sind automatisierte Tests *kleiner* Codeeinheiten, die **isoliert** getestet werden und weisen Aspekte von White-Box und Black-Box Tests auf (siehe Theorie über Testverfahren). Im Wesentlichen ist ein Unit Test ein Programm, das die öffentlichen Methoden einer Klasse aufruft und überprüft, ob die Ergebnisse den Erwartungen entspricht.

Vorteile von Unit Tests

Wenn Sie Ihre Komponententests richtig geschrieben haben, bringen sie mehrere Vorteile bezgl. der Wartung des Codes:

- **Dank Komponententests werden Fehler bereits während der Entwicklung gefunden:** Wenn Sie einen Codeabschnitt umgestalten oder erweitern und Ihre früheren Tests fehlschlagen, haben Sie das Verhalten dieser Methode geändert, was auf einen Fehler hinweisen könnte.
- Der Korrekturaufwand für neu gefundenen Fehler ist erheblich **geringer**, als wenn diese erst im laufenden Betrieb gefunden werden.
- Mit anderen Worten, Komponententests schützen Ihren Code vor **Regression** (d.h. Systemänderungen, die Fehler verursachen).
- Nachdem Sie Ihren Code geändert haben, **müssen** Sie die Testsuite ausführen! Auf diese Weise finden Sie heraus, ob es eine Regression gab (wenn ein Test fehlschlägt).

Unit Tests bilden die Basis der Testpyramide



Wie sind Unit Tests zu implementieren?

So wie es die SOLID Principles für die Programmierung im Allgemeinen haben, gibt es auch Prinzipien für allgemeine bewährte Verfahren beim Unit Testing: die **FIRST Principles**. Gehen wir sie durch.

[F]ast

Sie **sollten nicht** zögern, die Testsuite zu jedem Zeitpunkt des Entwicklungszyklus auszuführen, selbst wenn es Tausende von Unit Tests gibt. Sie sollten in Sekundenschnelle ausgeführt werden. Wenn die Ausführung eines Tests zu lange dauert, leistet dieser wahrscheinlich mehr, als er sollte – und ist daher kein **Unit** Test!

[I]ndependent resp. [I]solated

Jeder einzelne Test sollte **unabhängig** von allen anderen sein, damit seine Ergebnisse **nicht** von anderen Faktoren beeinflusst werden. Mit dieser Definition sollten Sie normalerweise den „3 A's des Testens" folgen: **Arrange, Act, Assert** (auch bekannt als „Given-When-Then").

- **Arrange:** Alle erforderlichen Daten sollten dem Test in diesem Abschnitt zur Verfügung gestellt werden und diese sollten **nicht** von Ihrer Umgebung abhängen.
- **Act:** Hier führen Sie die zu testende Methode aus.
- **Assert:** Unit Tests haben möglichst nur **ein** Ergebnis. Das bedeutet, dass pro Test nur **ein** bestimmter Zustand eines Objekts überprüft werden sollte. Dies bedeutet, dass sich **alle** getesteten Variablen auf die von Ihnen ausgeführte Methode beziehen sollten.

[R]epeatable

Tests sollten wiederholbar und deterministisch sein: d.h. das Ergebnis ist jedes Mal dasselbe und zwar unabhängig von der Umgebung.

[S]self-Validating

Der Test selbst sollte Ihnen sagen, ob er bestanden wurde. Eine manuelle Überprüfung der Werte sollte nicht erforderlich sein. Die meisten Bibliotheken (wie `pytest`, siehe Doku) arbeiten zugunsten dieses Prinzips.

[T]horough

Ihr Test sollte alle *„glücklichen Pfade"* einer Methode abdecken, alle Randfälle (bei denen Sie glauben, dass der Test fehlschlagen könnte), illegale Argumente, Sicherheitslücken, grosse Wert etc. Mit anderen Worten: jedes mögliche Szenario sollte getestet werden und nicht nur genug, um eine (nahezu) 100%ige Codeabdeckung zu haben.

Das zusätzliche T: Timely

Gemäss TDD sollten Komponententests **vor** dem zu testenden Produktionscode geschrieben werden. Dies geschieht, indem die Abstraktion (Schnittstelle) und dann der Test allein auf der Grundlage der Methodensignatur und -spezifikation geschrieben wird. Nachdem der Test geschrieben ist, kann der produktive Code implementiert und direkt getestet werden.

Credits: Der Inhalt dieses Artikels wurde grösstenteils direkt aus dem Englischen übernommen und nur leicht modifiziert. Das Original stammt aus diesem Beitrag der DEV Community, verfasst von Lucas Fonseca Mundim