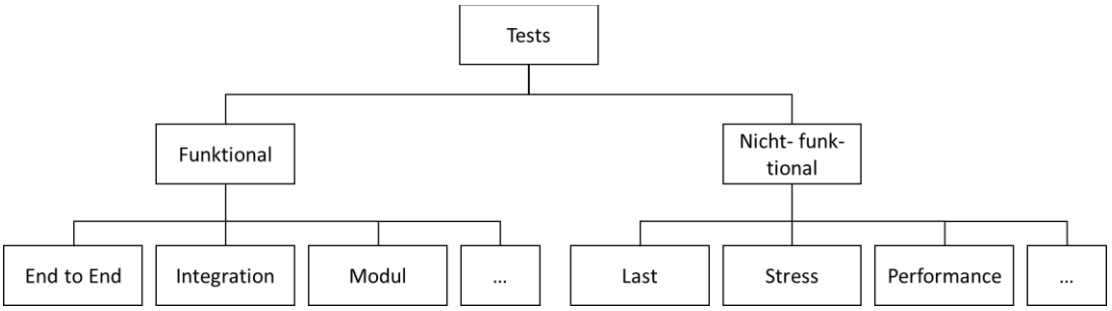
1 Definition Test

Ein Softwaretest soll prüfen, ob eine Software den definierten Anforderungen entspricht. Hierbei können wir grob zwischen zwei Testkategorien unterscheiden:



Testart:	**Ziel:**
	Prüfung der kleinsten Programmeinheit auf Funktionalität
	Prüfung auf das Zusammenspiel verschiedener SW-Elemente
	Test aus Sicht des Endnutzers (auch Applikationstest genannt)
	Prüfung, ob Software bei hoher Last (bspw. Nutzerzahl) noch funktioniert
	Prüfung, ob SW bei gezielter Fehlnutzung noch funktioniert.
	Prüfung, ob SW die geforderten Antwortzeiten erfüllt

Daneben gibt es noch weitere Testformen, welche prinzipiell jede erdenkliche Eigenschaft einer Software im Fokus haben können.

2 Schreibtischtest

Der Schreibtischtest erfordert ein manuelles Durcharbeiten des Codes mit Hilfe von „Stift und Papier“. Hierbei wird der innere Zustand der Funktion mit Hilfe er Variablen dargestellt.

```
…
function determinePrice(pricePerUnit, noOfUnits,
                        discountUnits, discount) {
  if(noOfUnits > discountUnits) {
    var discountFactor = 1-discount;
  }
  return discountFactor * pricePerUnit * noOfUnits;
}
…
```

pricePerUnit	noOfUnits	discountUnits	discount	discountFactor	Bedingung	return
4.0	15	10	0.1			

Ergänzen sie die Tabelle um die einzelnen Werte der Variablen, dem Ergebnis der (in diesem Code einzigen) Bedingungsprüfung und dem Rückgabewert.

Der Schreibtischtest spielt in der Praxis eine absolut untergeordnete Rolle. Zum einen, weil der Rechner den Code sehr viel effizienter abarbeitet und zum anderen, weil der Entwickler meist die potentiellen Denkfehler, welche er beim Programmieren begeht auch beim Schreibtischtest einfließen lässt – er ist also „vorbelastet“ und kann den Code nicht neutral bewerten.

3 Modultest

Im Modultest werden abgrenzbare Elemente eines Codes getestet. Dies sind meist einzelne Funktionen – können aber auch größere Codebereiche umfassen. Um einen solchen Test durchführen zu können, muss der Tester Einblicke in den Code haben.

Tests, in denen der Tester den internen Aufbau des zu testenden Programms kennt, nennt man:

Beim Modultest ist es wichtig, dass die Tests **voneinander unabhängig** und **wiederholbar** sind. Dadurch sind sie problemlos **automatisierbar**. Diese Anforderung führt jedoch zu folgenden Punkten:

- Ergebnisse von Funktionen hängen idealerweise ausschließlich von den Parametern ab („reine Funktion“)
- Wenn trotzdem Zugriffe auf globale Variablen (oder Instanzvariablen) notwendig sind, so müssen diese mit künstlichen Werten belegt werden („mocking“)
- Objekte, welche als Parameter übergeben werden, müssen ebenfalls mit künstlichen Werten belegt werden.
- Um eine sinnvolle Testüberdeckung zu erlangen, müssen meist eine relativ große Anzahl von Testfällen erzeugt werden.

Für die meisten Programmiersprachen gibt es Unittest Frameworks. Diese folgen meist einem einheitlichen Schema:

- Benennung des Testfalls
- Aufruf der Funktion mit fest definierten Testdaten („Parametern“) im Rahmen einer „Assertion“
- Die Assertion formuliert eine Annahme – meist, dass die Funktion einen gewissen Wert zurückgibt. Hier existieren aber noch weitere Möglichkeiten
    - Funktion gibt einen vorgegebenen Wert zurück
    - Funktion gibt einen vorgegebenen Wert nicht zurück
    - Funktion liefert ein true (oder false)
    - Funktion wirft eine definierte Exception
    - …
- Die Testfälle laufen in einer eigenen Applikation (bzw. eigenem Skript), welches am Ende die Anzahl (und Grund) der Fehler ausgibt

Ein für Java primär genutztes Framework wäre bspw. JUnit.

4 Integrationstest

Im nächsten Schritt erfolgt der Integrationstest. Hierbei wird das Zusammenspiel der vorher getesteten „Units“, bzw. größerer Einheiten geprüft. Hierbei spielen oft die Schnittstelen zwischen den Komponenten eine herausragende Rolle. Wie Unittests auch, sollten Integrationstests automatisierbar gestaltet werden. Eine wichtige Anforderung an den Integrationstest ist, dass man Testdaten benötigt, welche oftmals „künstlich“ erstellt werden müssen. Dies fordert mitunter einiges an Aufwand.

Integrationstest werden im Regelfall mit Wissen über die Implementierung durchgeführt, was sie zu einem Vertreter der Whitebox Tests macht.

5 End to End Test

Zum Schluss wird der End to End Test (oder auch Applikationstest) durchgeführt. Dieser wird aus Sicht des Nutzers durchgeführt und hat – wie der Name schon sagt – die gesamte Applikation im Fokus. Hier soll der Tester idealerweise keinerlei Einblicke in die eigentliche Implementierung haben. Tests, in denen der Tester den internen Aufbau des zu testenden Programms nicht kennt, nennt man:

End to End Tests benötigen oftmals „reale Daten“, welche meist aus den Produktivsystemen entnommen werden. Hierbei müssen aus Datenschutzgründen die Daten soweit anonymisiert werden, dass der Rückschluss auf die dahinterliegenden Personen nicht mehr möglich ist. Eine größere Herausforderung beim End to End Test ist meist die Testautomation. Bei Webseiten hilft oftmals eine klare Struktur des HTML DOM (Document Object Model), so dass mit Hilfe von Tools wie

bspw. Cypress eine programmierte Simulation eines Nutzers erfolgen kann. Bei anderen GUI Clients müssen hier ggf. kostenpflichtige Programme genutzt werden.

6 Überdeckungsmaß

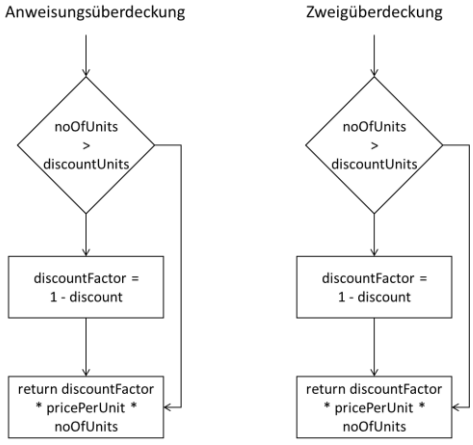
Unter Überdeckungsmaß versteht man die Angabe, welcher Anteil des Codes bei einer Anzahl zusammenhängender Testläufe vom Rechner durchlaufen wurde. Hierbei ist der Begriff „Code" jedoch interpretierbar, weshalb man den Code als einen „Kontrollfluss" interpretiert und man hier auch von kontrollflussbasierten Testen spricht. Man unterscheidet hier verschiedene Überdeckungsmaße:

Testart:	**Ziel:**
	Welcher Anteil an Codezeilen wurde verarbeitet.
	Welcher Anteil an Anweisungen wurde verarbeitet.
	Welcher Anteil an Verzweigungen (true/false – Fälle) wurde verarbeitet.
	Welcher Anteil an Bedingungen wurde verarbeitet.

Darüber hinaus gibt es noch die Pfadüberdeckung, welche jedoch in der Praxis keine große Rolle spielt. Die Zeilenüberdeckung ist kein sinnvolles Maß, da in den meisten Programmiersprachen in einer Zeile mehrere Anweisungen stehen können. Für die visuelle Darstellung in IDEs bietet sich die Zeilenüberdeckung jedoch an.

Wichtig sind die Unterscheidung der Anweisungs- und Zweigüberdeckung. Den Unterschied sieht man im Programmablaufplan PAP (als bekannteste Darstellung eines Kontrollflusses) am besten. Anweisungsüberdeckung fokussiert sich auf die flächigen Elemente des PAP, in denen Anweisungen und Bedingungen eingetragen werden. Zweigüberdeckung wiederum fokussiert auf die einzelnen Zweige – sprich die Pfeile des PAP.

Markieren Sie rechts farbig die relevanten Bereiche für die Anweisungs- bzw. Zweigüberdeckung.



Eine weitere wichtige Größe stellt die Bedingungsüberdeckung dar. Hierbei wird ein weiterer Fokus auf die Bedingung gelegt, welche in Verzweigungen und Schleifen zu finden sind und durch mehrere Boolsche Ausdrücke zusammengesetzt werden können. Folgendes Java Beispiel soll dies verdeutlichen:

```
if (sEingabe.equals("y") > 0 && myArray.length < 100)
```

Eine Eingabe mit `"n"` Bedingungsüberdeckung bedeuten, da die Prüfung `myArray.length < 100` nicht durchgeführt werden würde. Wenn bspw. das Array `null` wäre, so würde nur bei `"y"` eine `NullPointerException` geworfen werden. Nun ist der oben gezeigte Fall, dass `sEingabe` den Wert `"y"` aufweist, „lediglich" eine einfache Bedingungsüberdeckung. Besser wäre eine mehrfache Bedingungsüberdeckung, bei der alle möglichen true/false Kombinationen abgeprüft werden. Im folgenden Fall würde eine reine `true/true` Prüfung das Problem des Codes nicht aufdecken:

```
if (myArray.length > 0 && myArray != null)
```

Erst mit `false/false` würden wir sehen, dass wir hier in eine `NullPointerException` hineinlaufen würden.

7 Datenauswahl

Neben der kontrollflussorientierten Sichtweise können wir noch eine Datensichtweise ansetzen. Hierbei sehen wir uns die Daten an, mit denen wir testen wollen. Vor allem die „edge Cases" spielen hier eine große Rolle. Wenn wir bspw. bei folgendem Code nur bspw. die 5 und die 15 für die Variable noOfUnits einsetzen, so stellen wir nicht fest, ob das Verhalten bei genau 10 den Erwartungen entspricht, oder nicht:

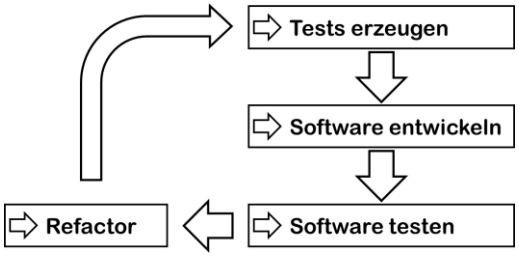
```
if (noOfUnits > 10)
```

Grundsätzlich gilt, dass wir möglichst viele unterschiedliche Werte (bspw. auch negative, wenn vom Datentyp her möglich Rationale Zahlen etc.) getestet werden sollten. Es sollte jedoch auch klar sein, dass wir niemals alle Werte (und schon gar nicht Wertekombinationen) testen können. Insofern ist folgender Satz wichtig:

Ein Test kann niemals eine 100% Fehlerfreiheit garantieren, sondern nur das Funktionieren der Testfälle!

8 Test Driven Development

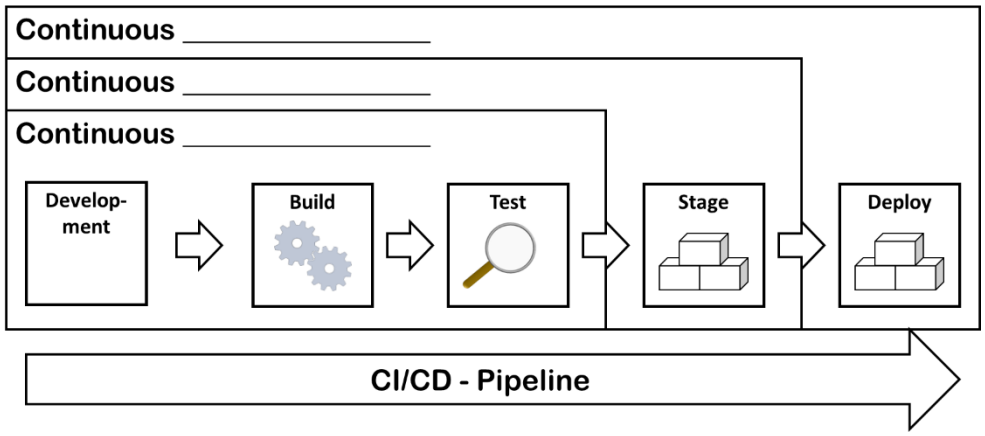
Unter TDD versteht man die integrale Eingliederung von (automatisierten) Tests in den Entwicklungsprozess. Hierbei werden oftmals die Test vor der eigentlichen Entwicklung erzeugt. Dadurch kann ein geschlossener Kreis für die Entwicklungsphase entstehen:



Der nächste Schritt bei der Automation ist die Konzentration auf den gesamten Prozess von der Entwicklung bis zum Deployment auf den Produktionsserver

9 Continuous Integration

Um die Anzahl der möglichen Fehler beim Build (und ggf. auch beim Test) zu minimieren, möchte man möglichst zeitnah die Entwicklungsergebnisse in den Test überführen. Hierzu muss die Prozesskette Development (inkl. Versionsmanagement) -> Build -> Test möglichst automatisiert sein und man spricht hier von „Continuous Integration“. Wird die Vorbereitung für die Produktionseinführung ebenfalls automatisiert, nennt man dies „Continuous Delivery“ und der vollautomatische Prozess bis zur Produktionseinführung ist das „Continuous Deployment“:



Die einzelnen Schritte werden in einer „Pipeline“ aneinandergehängt. Tools wie bspw. „Jenkins“ dienen dazu, diesen automatisierten Prozess zu unterstützen.

Die Einbindung des Operations in diesen Prozess – bspw. durch einen definierten Feedbackkanal von der Produktion in die einzelnen vorgelagerten Schritte bezeichnet man als „DevOps“