Agiles Testen

Das agile Testen ist ein Ansatz, bei dem die Testaktivitäten in den gesamten Entwicklungsprozess integriert werden.

Test Driven Development (TDD)

TDD ist eine Methode, bei der zuerst die Tests geschrieben werden und dann der Code. Zuerst wird die grundlegende Code-Struktur definiert, dann wird ein Test geschrieben, der diese Struktur überprüft. Der Test wird dann ausgeführt und sollte fehlschlagen, da der Code noch nicht implementiert wurde. Dann wird der Code implementiert, um den Test zu bestehen. Das Code wird in kleinen Schritten implementiert, wobei nach jedem Schritt ein Test geschrieben wird, dieser zuerst fehlschlagen sollte und dann der Code implementiert wird, um den Test zu bestehen.

Regeln von Unit-Tests

Atomar
Ein Unit-Test sollte nur eine Sache testen.

Schnell
Unit-Test sollten so schnell wie möglich ausgeführt werden können, damit sie oft wiederholt werden können,ohne Zeit zu verschwenden.

Isoliert
Es ist wichtig, dass die Ergebnisse eines Unit-Tests nicht von anderen Tests abhängen.

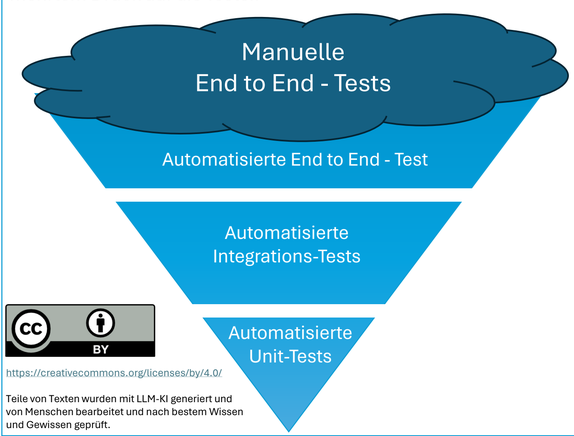
Deterministisch/Wiederholbar
Ein Unit-Test sollte immer das gleiche Ergebnis liefern, wenn er auf dem gleichen Code ausgeführt wird.

Einfach
Tests sollten einfach sein, damit sie zuverlässig funktionieren, leicht zu verstehen und leicht zu warten sind.

Lesbar
Ein Unit-Test sollte leicht zu lesen und zu verstehen sein, damit er leicht gewartet werden kann.

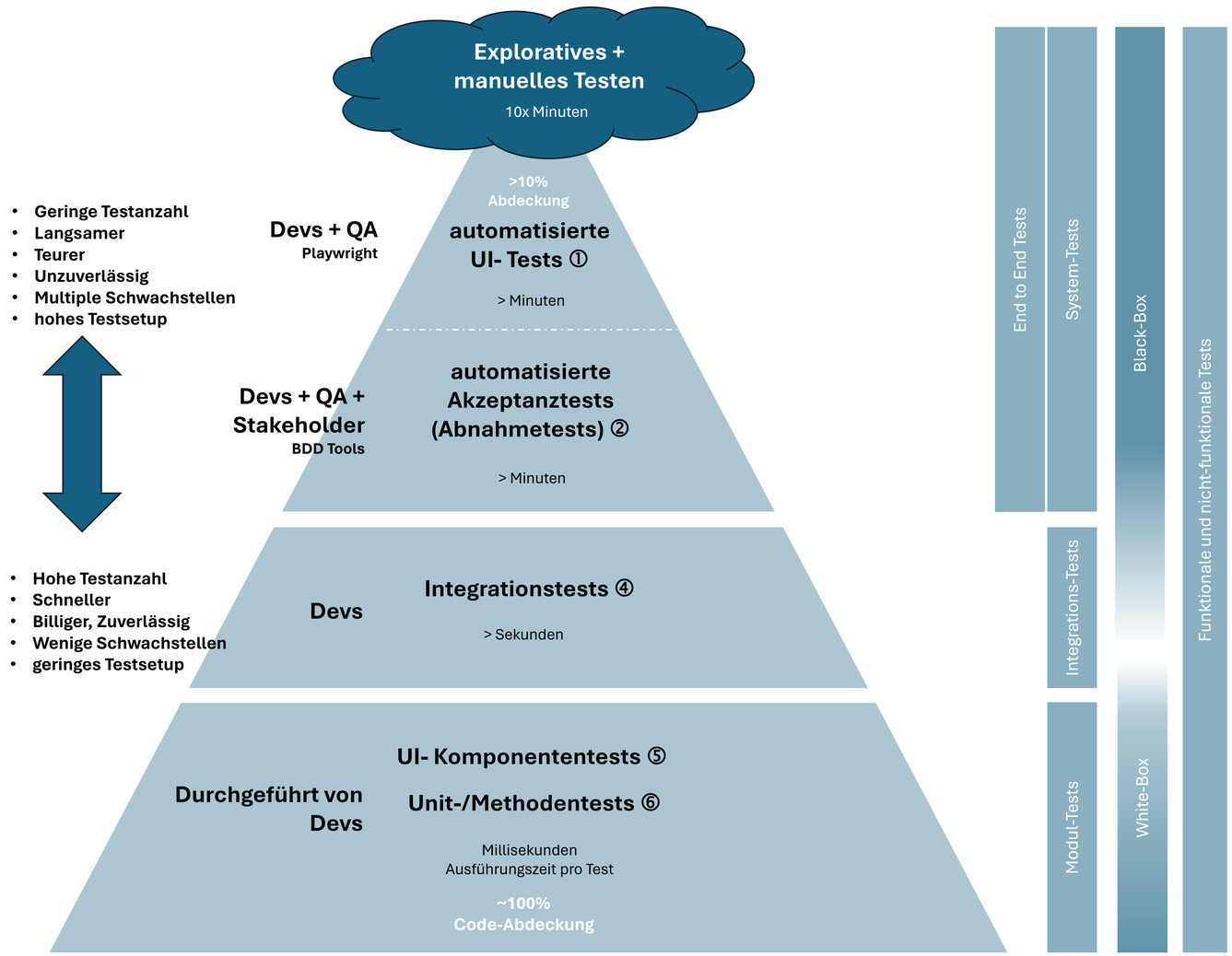
Ice Cream Cone Antimuster

Das Ice Cream Cone Antimuster beschreibt eine Situation, in der die Anzahl der manuellen Tests die Anzahl der automatisierten Tests übersteigt. Diese weitverbreitete Praxis kann Unternehmen vor ernsthafte Herausforderungen bei der Software-Wartung und der Agilität stellen. Der Mangel an ausreichender Testautomatisierung führt zu langsameren Release-Zyklen, höheren Kosten und vermehrtem Druck auf die Tester.



Die Testpyramide

Die Testpyramide sollte als Empfehlung für die Verteilung von Tests in einem Softwareprojekt gesehen werden, und passt nicht immer in jeden Kontext.



- **Geringe Testanzahl**
- **Langsamer**
- **Teurer**
- **Unzuverlässig**
- **Multiple Schwachstellen**
- **hohes Testsetup**

- **Hohe Testanzahl**
- **Schneller**
- **Billiger, Zuverlässig**
- **Wenige Schwachstellen**
- **geringes Testsetup**

Test-Frameworks

Java
JUnit, TestNG
JavaScript
Jest, Mocha, Cypress
BDD
AssertThat (Jira),
Cucumber,
Fitnesse,
JBehave,
JDave,
Concordion

SCAN ME

Behaviour Driven Development (BDD)

Behavior Driven Development ist eine Entwicklungsmethode, die die Kommunikation zwischen Entwicklern, Testern und Geschäftsanalysten verbessert, indem sie eine gemeinsame Sprache für die Spezifikation von Softwareverhalten nutzt. Es verwendet "User Stories" mit akzeptierbaren Kriterien, die in Tests umgewandelt werden, bevor die eigentliche Entwicklung beginnt. Diese Tests dienen als lebendige Dokumentation und sind in der natürlichen, für alle verständlichen "Gherkin"-Syntax verfasst. BDD fördert die Kollaboration und integriert sich nahtlos in agile Praktiken, was die Entwicklungszyklen effizienter macht. Ziel ist es, sicherzustellen, dass die entwickelte Software genau den Geschäftsanforderungen entspricht.

White-Box-Tests

Mit dieser Klasse von Tests werden Tests beschrieben, die auf der Kenntnis des Codes basieren.

Black-Box-Tests

Mit dieser Klasse von Tests werden Tests beschrieben, die auf der Kenntnis der Spezifikation basieren, d.h. die Tests werden ohne Kenntnis des Codes geschrieben, sondern nur auf Basis des Verhaltens der Software.

Funktionale Tests

Unit-Tests ⑥
Diese Art von Tests prüfen grundlegende Elemente der Software in Isolation und mit innerem Detailwissen über den Code. Sie sind die wichtigsten Tests wenn es darum geht sicherzustellen, dass die Software korrekt funktioniert. Allgemein werden Unit-Tests auch Modultests oder Komponententests genannt. Sie sind schnell und einfach zu schreiben sowie auszuführen.

UI-Komponententests ⑤
UI-Kompontententests prüfen einzelne entwickelte Komponenten der Benutzeroberfläche, indem sie ihr Verhalten isoliert oder in Verbund mit anderen Komponenten prüfen.

Integrationstests ④
Integrationstests prüfen grundlegende Elemente, wobei einige Infrastrukturkomponenten wie Datenbanken oder Netzwerke eingebunden werden. Ihre Abhängigkeiten können zu Störungen in der Ausführung führen.

Systemtests
Diese Art von Tests, auch genannt End-to-End-Tests, prüfen viele verschiedene Elemente der Softwarearchitektur, indem sie das Verhalten eines Benutzers simulieren.

Akzeptanztests ②
Diese Art von Tests prüfen, ob die Software die Anforderungen des Kunden erfüllt.

Automatisierte UI-Tests ①
Diese Testart ist Teil der Modultests und wird auch als Komponententest bezeichnet. Sie prüfen die korrekte Funktionalität der Benutzeroberfläche, jedoch ohne die Anbindung an die Umgebung.

Regressionstests
Diese Art von Tests prüfen, ob Funktionalität die bereits implementiert wurde, auch weiterhin funktioniert, nachdem neue Funktionalität hinzugefügt wurde. Kurzgesagt, jeder funktionale Test ist ein Regressionstest, sofern er bei einer getesteten Änderung am Code fehlschlägt.

Nicht funktionale Tests

Lasttests
Diese Tests prüfen, wie die Software auf eine hohe Anzahl von Anfragen reagiert.

Leistungstests
Diese Art von Tests prüfen, wie schnell die Software auf Anfragen reagiert.

Usability-Tests
Diese Art von Tests prüfen, wie einfach die Software zu bedienen ist.

Sicherheitstests
Diese Art von Tests prüfen, ob die Software gegen bekannte Angriffe von außen geschützt ist.

Kompatibilitätstests
Diese Art von Tests prüfen, ob die Software auf verschiedenen Plattformen und Geräten funktioniert.

Komponenten (lat. Zusammengesetzt)

In der Softwareentwicklung bezeichnet der Begriff "Komponente" eine modulare Einheit mit spezifischen Funktionen, die unabhängig entwickelt, getestet und wiederverwendet werden kann. Diese Komponenten interagieren über definierte Schnittstellen miteinander, was die Wartbarkeit und Skalierbarkeit von Software verbessert. Häufig bezieht sich "Komponente" speziell auf GUI-Elemente wie Buttons oder Textfelder, was zu Verwechslungen führen kann.