XCEPTANCE
Software Technologies

Software-Qualitätssicherung, Teil III: Testautomatisierung

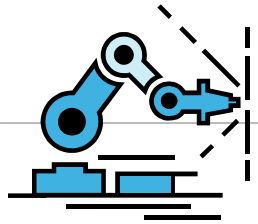
Friedrich-Schiller-Universität Jena, Wintersemester 2017/2018

Ronny Vogel, Xceptance GmbH

Agenda

- Testautomatisierung – Grundlagen
- Testautomatisierung – Ebenen
- Grenzen der Testautomatisierung
- TDD, BDD, ATDD

Testautomatisierung – Begriff

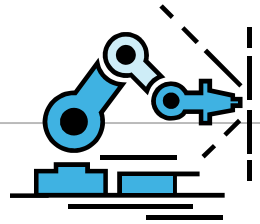


Was ist Testautomatisierung?

- Automatisierung von Aktivitäten im Test mit geeigneten Werkzeugen, z.B.:
  - Testdatengenerierung
  - automatisierte Testausführung nach jedem Build
  - Erzeugen von Testdokumentation mit Dokumentgeneratoren
  - automatisierte Testfallerstellung (Modellbasiertes Testen)
  - Lasttests

- Begriff meist jedoch im engeren Sinne gebraucht, so auch im Weiteren
  ➔ automatisierte Ausführung funktionaler Tests

Testautomatisierung – Ziele

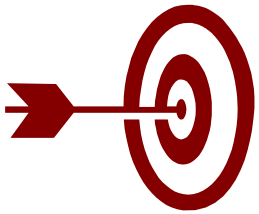


Höhere Qualität durch

- häufigere Ausführung umfassender Testsuiten
- umfassenden Test auch bei kleinen Änderungen oder kleinen Releases
- Testausführung mit einer Anzahl verschiedener Konfigurationen
- exakt reproduzierbare Wiederholung von Testschritten
- verringertes Restrisiko

Zeitersparnis

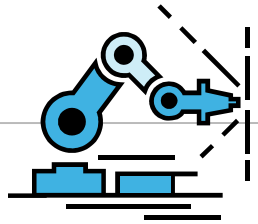
- schnellere Testdurchführung, früheres Feedback
- destabilisierende Änderungen und Regression schneller erkennen
- Voraussetzung für eine effiziente QS in agilen Vorgehensmodellen!



Kostenreduzierung

- weniger wiederkehrender manueller Testaufwand

Testautomatisierung – Vorteile (1)



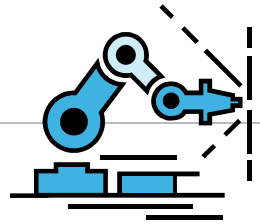
Befreit Tester von monotonen, wiederholten Tätigkeiten

- mehr Zeit für interessantere, wichtigere Aufgaben
- Konzentration auf anspruchsvollere Testszenarien, exploratives Testen
- höherer kreativer Anteil ➔ Motivation
- mehr Zeit für tiefere Testabdeckung

Automatische Ausführung

- unbeaufsichtigte Testausführung möglich
- unabhängig von der Verfügbarkeit manueller Tester
- in Continous Integration einbindbar
- Ergebnisse liegen schneller vor
- weniger fehleranfällig
- automatische Protokollierung

Testautomatisierung – Vorteile (2)



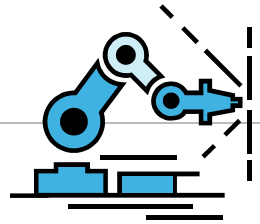
Schnell und häufig wiederholbar

- verkürzt Entwicklungs- und Releasezyklen
- erhöht das Vertrauen in die gelieferte Qualität
- „Sicherheitsnetz“ bei Refactoring und anderen Änderungen

Höhere Testabdeckung in wiederholten Tests

- kontinuierlicher Ausbau der Testfallanzahl bei nur minimal steigendem Ausführungsaufwand
- Prüfung vieler Varianten möglich (datengetriebene Tests)
- Fälle abdeckbar, die manuell nicht effizient getestet werden können
- Beispiele:
  - Prüfzifferberechnung von Kontonummern mit etwa 100 Verfahren
  - Crawling über eine größere Website zur Sicherstellung der Datenqualität; Alle Links funktional? Alle Bilder vorhanden? Grundelemente auf jeder Seite vorhanden (AGB, Impressum)?, …

Testautomatisierung – Vorteile (3)



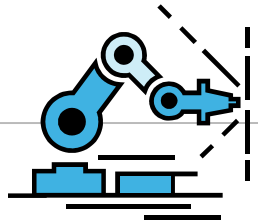
Bessere APIs, besserer Code

- Testautomatisierung ist oft die erste Nutzung von APIs
  ➔ Feedback zur API-Verbesserung
- bei Test-Driven Development (TDD) wird Testcode noch vor dem eigentlichen Code geschrieben
  ➔ Mängel schon bei Implementierung sicht- und behebbar
- Testautomatisierung auf UI-Ebene erzwingt oft Änderungen für Testbarkeit
  ➔ z.B. besserer HTML- und CSS-Stil bei Web-Anwendungen

Bessere Sichtbarkeit der Ergebnisse

- automatische Bereitstellung der Ergebnisse, z.B. auf Webseiten
- automatische Erstellung von Trendreports

Testautomatisierung – Nachteile



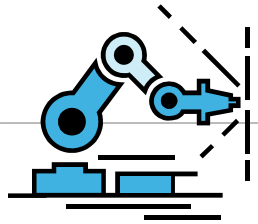
Aufwand

- hoher Initialaufwand
- permanente Pflege nötig
- Testcode ist Code
  - enthält Defekte
  - erfordert auch Review und Test
  - erfordert gelegentliches Refactoring
- Schulungsbedarf

Sonstiges

- teilweise komplexe Technologien und Werkzeuge
- Erstellung wartbarer, robuster automatisierter Testfälle ist anspruchsvoll
- erweitertes Wissen und Erfahrung erforderlich
- teilweise hohe Lizenzkosten

Testautomatisierung – Hinweise (1)

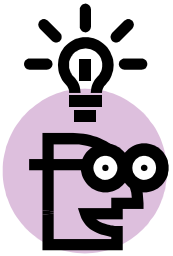


Keep it simple!

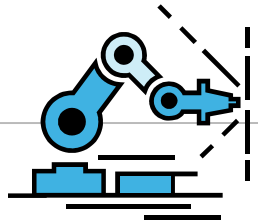
- mit einfachen, geschäftskritischen Fällen beginnen
- zunächst Erfahrungen sammeln, dann vom Einfachen zum Komplexen
- Testfälle klar, verständlich, leicht ausführbar gestalten
- komplexe Testsuite-Architekturen vermeiden
- nicht 100% Automatisierung versuchen; nur wo sinnvoll

Testfälle unabhängig voneinander gestalten!

- benötigte Vorbedingungen durch Testfall selbst oder in Setup-Methoden anlegen
- Ausführung in beliebiger Reihenfolge ermöglichen
- Fehlschlagen eines Testfalls darf keine Auswirkung auf andere haben
- wenn nötig, durch Tear-Down-/Clean-Up-Methoden Ausgangszustand wiederherstellen

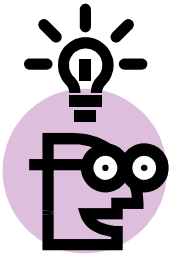


Testautomatisierung – Hinweise (2)



Allgemeine Hinweise

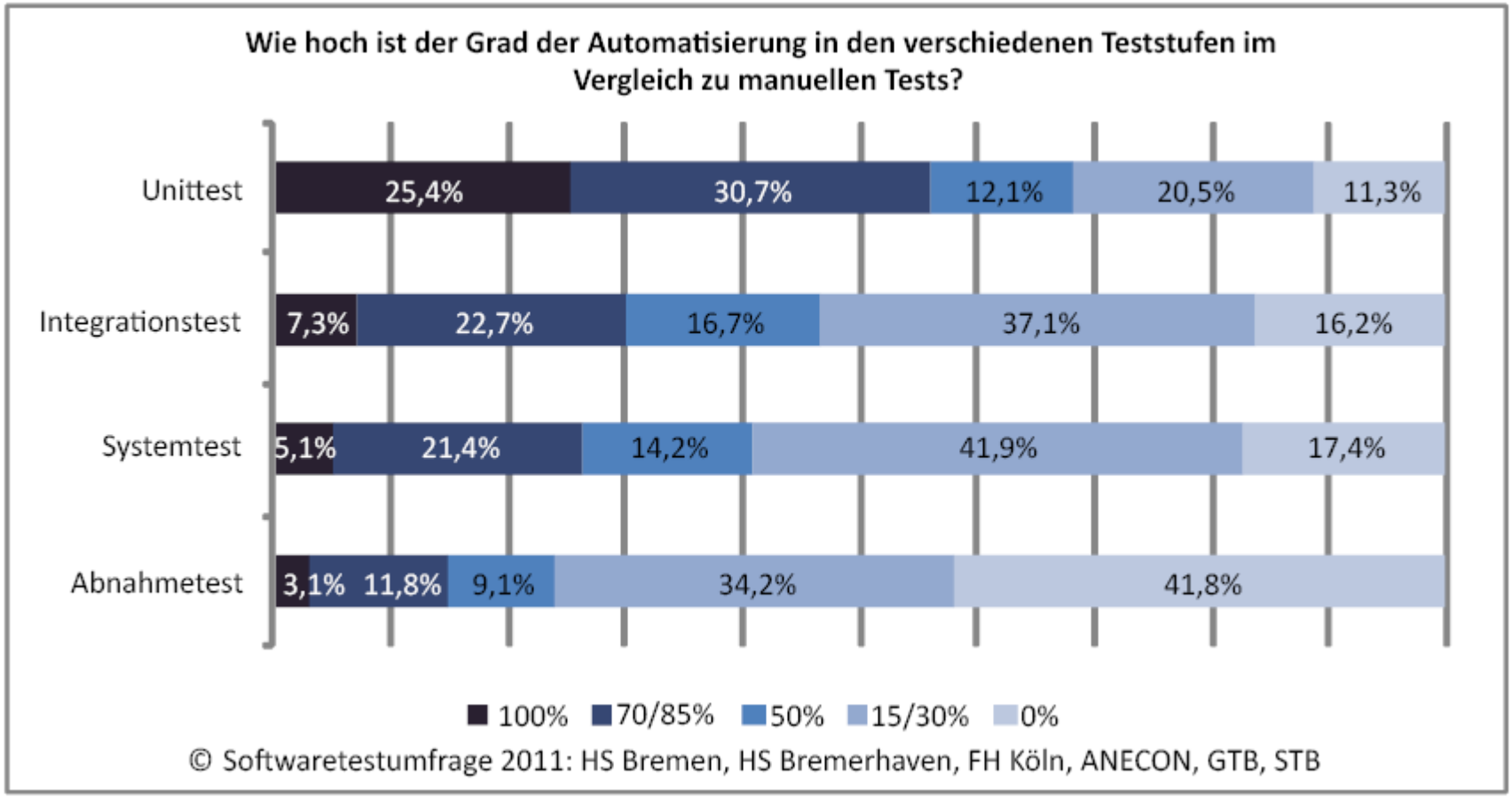
- vor Automatisierung von Testfällen immer manueller Test nötig/sinnvoll
- Testfälle möglichst atomar
- Testdaten auslagern, nicht hart codieren
- auf Robustheit der Testfälle achten
- Testfälle mit Produktcode versionieren
- auf möglichst geringe Laufzeit achten
- schnelle und langsame Testfälle nicht mischen, eventuell separate Testsuiten
- permanente Pflege und Erweiterung einplanen



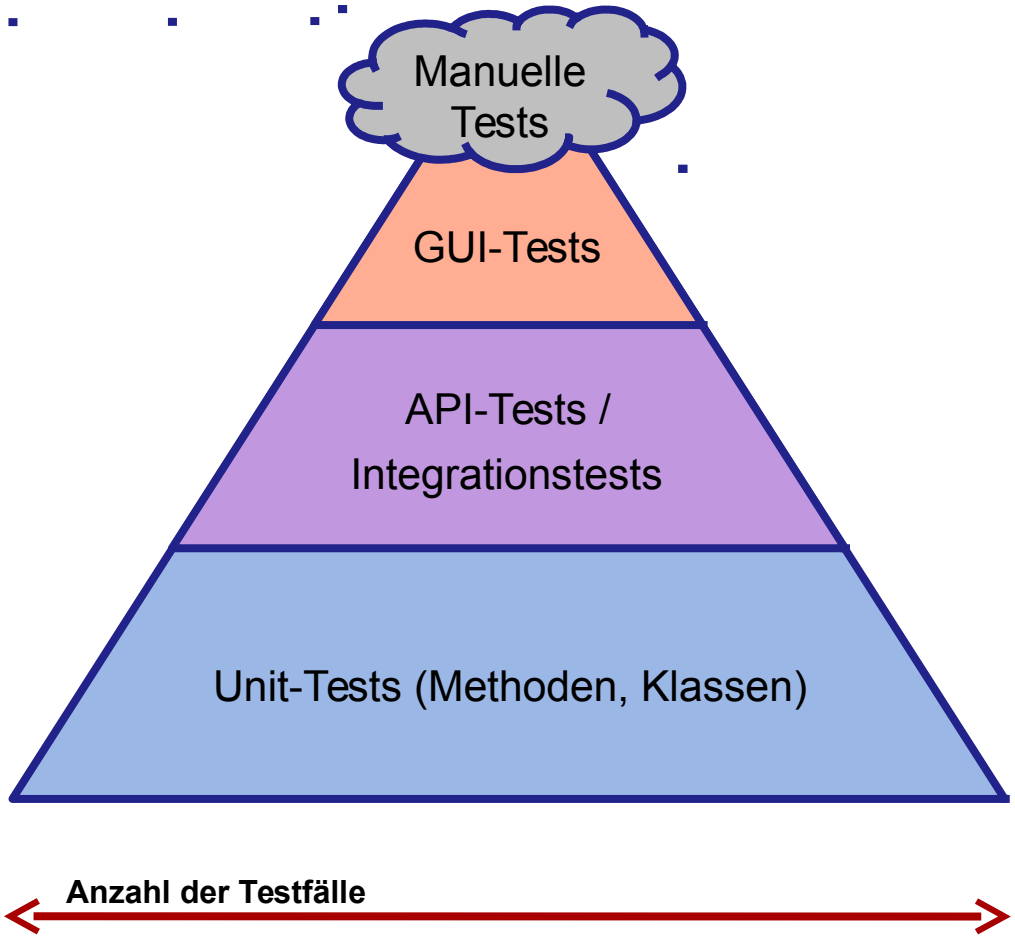
Agenda

- Testautomatisierung – Grundlagen
- Testautomatisierung – Ebenen
- Grenzen der Testautomatisierung
- TDD, BDD, ATDD

Ebenen der Testautomatisierung (1)

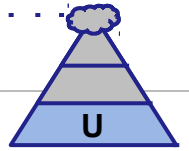


Ebenen der Testautomatisierung (2)



Testpyramide nach Mike Cohn:
"Succeeding with Agile", 2009

Ebene Unit-Tests (1)



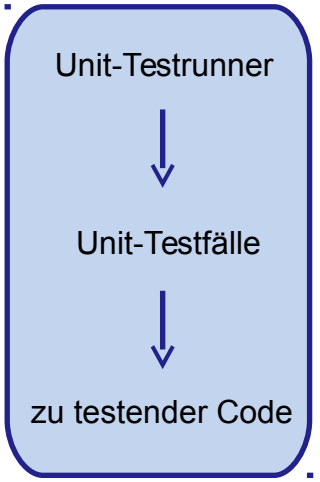
Erstellung (durch Entwickler selbst)

- je Testfall wird eine Testmethode in der jeweiligen Programmiersprache erstellt
- Testmethoden werden gekennzeichnet, z.B. mittels Annotationen:

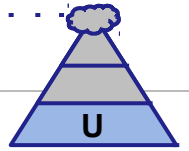
```
@Test
public void searchCustomersWithMultipleWildcards()
{…
     // Aufruf des zu testenden Codes
     …
     // Prüfung der Ergebnisse mit assert(…)-Konstrukten
     assert(…);
}
```

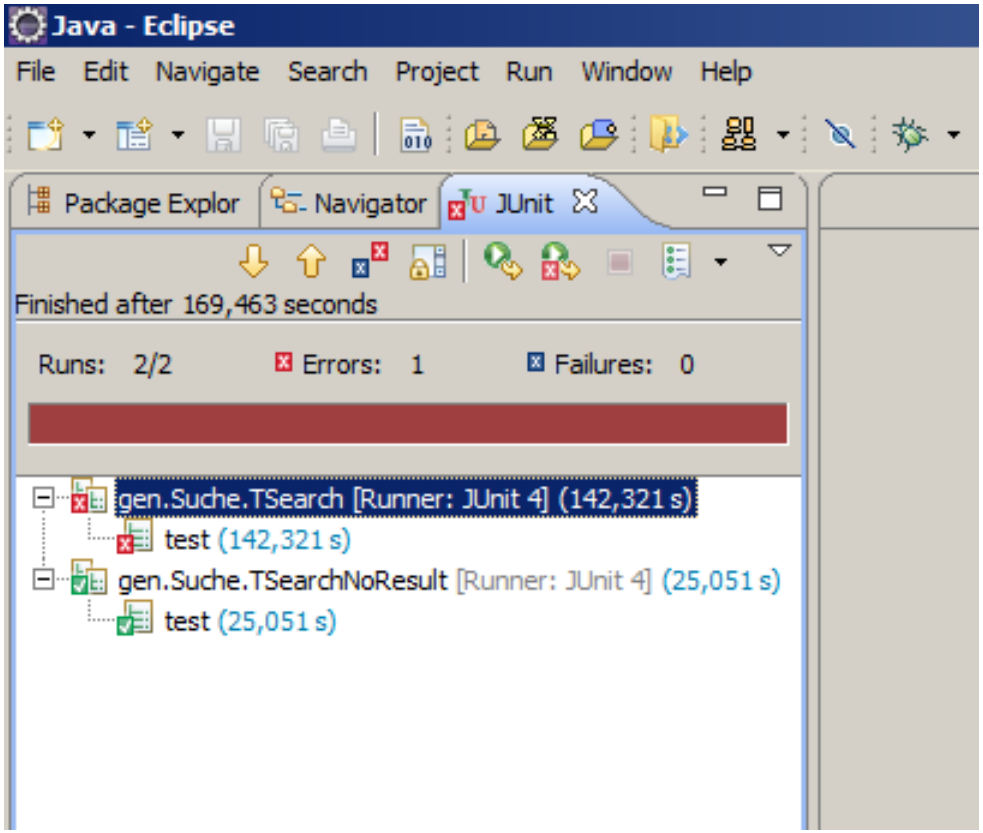
Ausführung

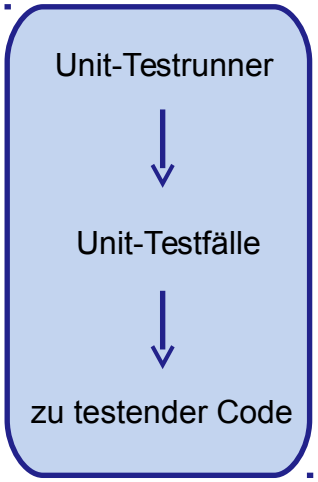
- trifft eine per assert() definierte Annahme nicht zu, wird ein Fehler über eine Exception an den Testrunner kommuniziert
- ein Unit-Testrunner erkennt die Testmethoden, führt sie aus und erstellt einen Ergebnisreport
- Testrunner als GUI-Anwendung oder kommandozeilenbasiert



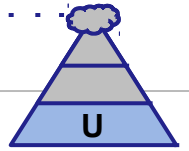
Ebene Unit-Tests (2)





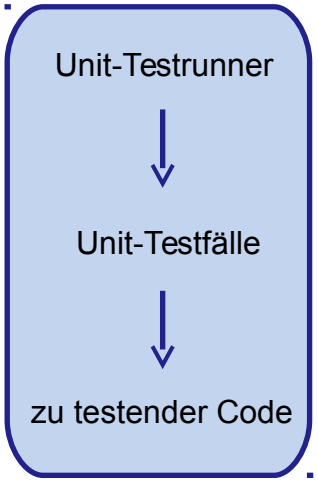


Ebene Unit-Tests (3)

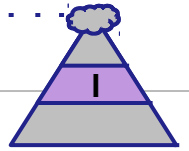


Merkmale

- testen Methoden/Funktionen der Anwendung (Code-Ebene)
- Fokus ist technische Korrektheit der Implementierung
- sollen sehr schnell und lokal begrenzt arbeiten, daher oft
  - ohne Datenbank-Kommunikation
  - ohne Netzwerkverbindungen
  - ohne Arbeit mit dem Dateisystem
  - …
- Nachbildung benötigter Komponenten durch „Test Doubles“
  ➔ „Mocking“-Frameworks
- bei Code-Änderung sofort auch Änderung des Testcodes
- führt Entwickler bereits vor Commit in zentrales Repository aus
- tragen massiv zur Verbesserung der Codequalität bei
- hohe Anzahl, oft zehntausende



Ebene Integrations- und API-Tests (1)



Tests für lokales API

- testen Zusammenspiel von Komponenten; Geschäftsfunktionen
- testen technische Korrektheit und korrekte Umsetzung von Anforderungen
- technologisch auch als Unit-Tests oder ähnlich realisierbar
- ebenfalls lokale Aufrufe von Code im gleichen Prozess
- Tests komplexer, höhere Funktionalität

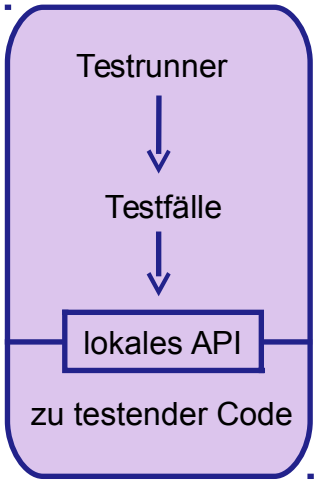
Vorteile:

- testen im Idealfall ein wohldefiniertes API
- über den Wartungszeitraum der Software wesentlich stabiler
- besser dokumentiert

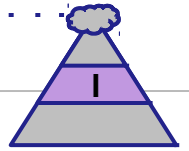
Nachteil:

Nur die Funktionen sind testbar…

- ...die im API zur Verfügung gestellt werden
- ...für die Setup und Ergebnisprüfung über das API möglich sind

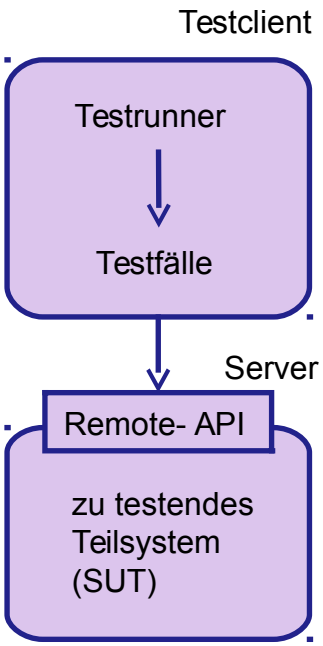


Ebene Integrations- und API-Tests (2)



Tests für Remote-API

- analog zum Test über lokales API, aber getrennte Prozesse
- synchrones API oder asynchrone Schnittstelle
- Testclient kommuniziert über bestimmte Protokolle mit SUT
- unterschiedliche Client-Technologien und Programmiersprachen von Testcode und SUT möglich
- Beispiel:
  - Remote Services über HTTP, REST-API
  - Messaging-Schnittstelle, z.B. Websphere MQ



Vorteile:

- wie bei lokalem API

Nachteile:

- wie bei lokalem API
- Testumgebung und Testfälle komplexer, oft langsamer

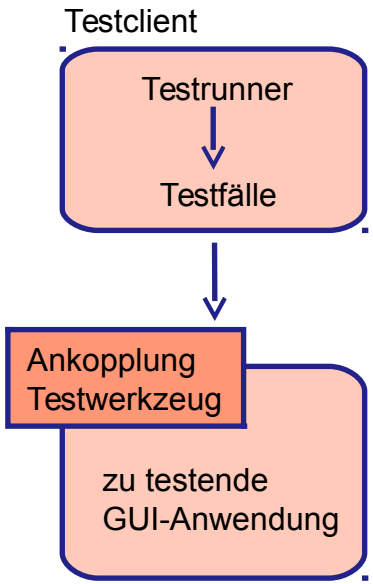
Ebene GUI-Tests, lokal

GUI-Tests für lokale Anwendungen („Fat Clients“)

- Test der Anwendung durch das GUI (nicht Test des GUI allein)
- Anwendersicht auf ein möglichst vollständiges Gesamtsystem
- Werkzeug koppelt sich an Anwendung oder Betriebssystem; versucht, GUI-Objekte und deren Manipulation zu erkennen, aufzuzeichnen und später nachzubilden

Varianten

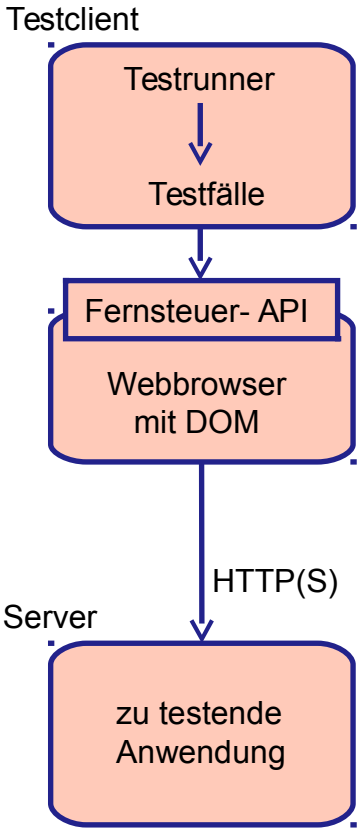
- Maus- und Tastaturaktionen koordinatenbasiert aufzeichnen und beim Testlauf wieder nachbilden
  ➔ schwierig, unflexibel, fehleranfällig, wenig robust
- Oberflächenelemente und Ereignisse vom darunter liegenden Fenstersystem erfassen
  ➔ schwierig, oft fehleranfällig
- über ein in das GUI integriertes Fernsteuerungs-API
  ➔ technisch ideal, aber Funktionalität oft eingeschränkt



Ebene GUI-Tests, Web-Anwendungen

GUI-Tests für Web-Anwendungen (browserbasiert)

- Sonderform des automatisiertenTests auf GUI-Ebene
- Webbrowser heute oft als universeller „Thin-Client“ genutzt
- durch Browser-Features oder Browser-Erweiterungen relativ gut umsetzbar
- arbeitet oft über DOM-/HTML-Zugriffe: Werkzeug erkennt und verändert DOM-Elemente im Browser (*DOM = Document Object Model*, standardisierte Schnittstelle mit Zugriffsmethoden auf die HTML-Struktur)
- viele existierende Werkzeuge, gut unterstützt
- Vorteil: Technologien und Erfahrungen mit Browser-Automatisierung breit verfügbar
- nutzt ferngesteuerten echten oder simulierten Webbrowser
- Beispiel: Selenium WebDriver als Fernsteuer-API steuert z.B. Firefox oder HtmlUnit („Headless Browser“ ohne GUI)



Agenda

- Testautomatisierung – Grundlagen
- Testautomatisierung – Ebenen
- Grenzen der Testautomatisierung
- TDD, BDD, ATDD

Grenzen der Testautomatisierung (1)

Aufwand

- Automatisierung auf höheren Ebenen komplex und daher oft teuer
- anfällig gegenüber Änderungen an der zu testenden Anwendung
  ➔ ohne permanente Pflege sterben Testfälle aus
- Aufwand versus Nutzen bei Erstellung und Pflege kritisch
  ➔ Testumfang gut auswählen und gegebenenfalls nachjustieren

Fehlende Intelligenz

- vergleichsweise schmalbandige Validierung der Ergebnisse
- prüft weniger Eigenschaften, als ein Tester; „Blick zur Seite“ fehlt
- erkennt keine GUI- und Layout-Fehler
- entwickelt sich nicht selbst weiter
- kann Abweichungen vom Soll-Ergebnis nicht bewerten (Testorakel-Problem)
- findet keine Fehler außerhalb vordefinierter Szenarien
- James Bach: „ein automatisierter Test testet nicht, er prüft nur“

Grenzen der Testautomatisierung (2)

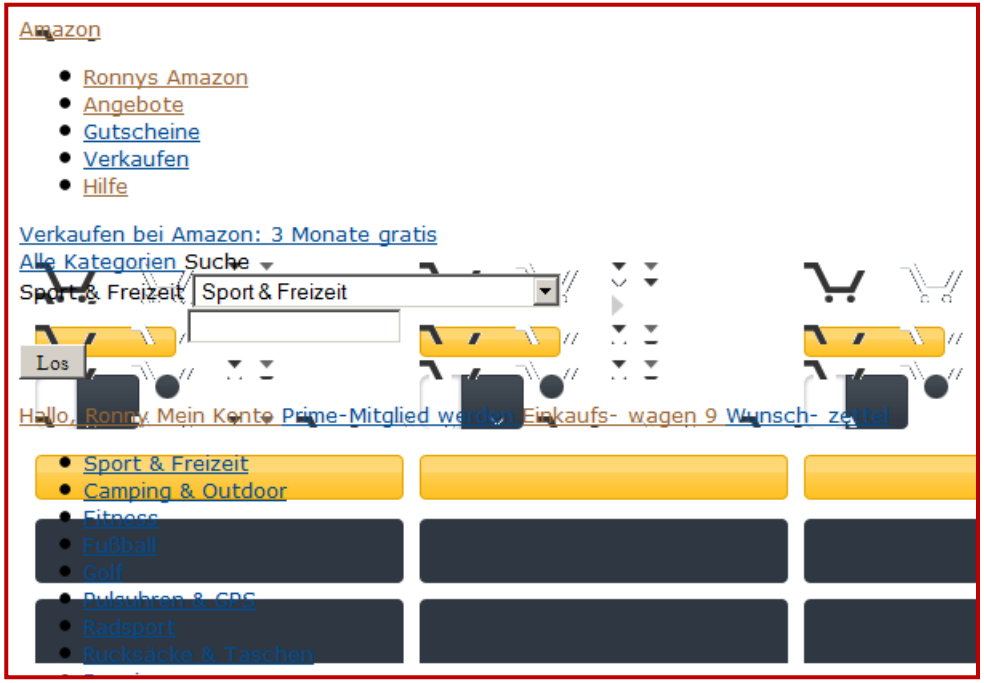
Sonstige Grenzen

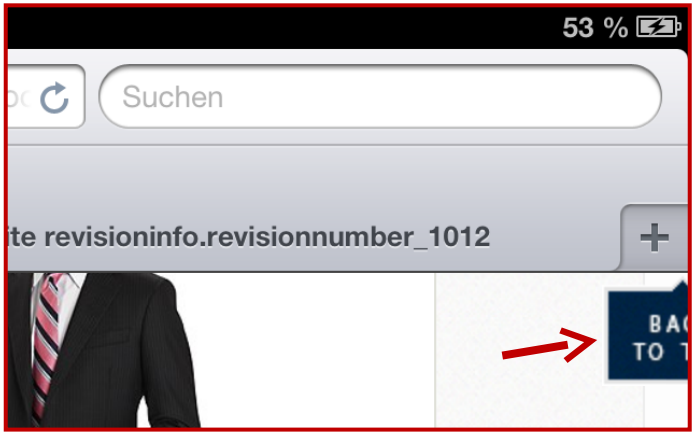
- manche Qualitätsmerkmale sind nicht sinnvoll automatisiert prüfbar
- Immunität gegen bisher übersehene Defekte
  - viele Erkenntnisse und gefundene Defekte bei Testfallerstellung
  - keine neuen Erkenntnisse durch wiederholte Testausführung
- oft eher technische Sichtweise
- Unit-Tests testen nur, was implementiert und wie es verstanden wurde…
- kein Test ohne Testfälle möglich - nur der aktuell implementierte Testumfang wird getestet

➔ **Betätigungsfeld unwillkürlich dort, wo gut automatisierbar**

➔ **Testautomatisierung wiegt Beteiligte oft in falscher Sicherheit**

Grenzen der Testautomatisierung (3)





Grenzen der Testautomatisierung (4)

« Zurück zum Suchergebnis

WEITERE BILDER

Zahnputzbecher

zu allen Artikelinformationen und Serviceleistungen

NEU im Sortiment
~~€ 17,90~~ (Sie sparen € 10,40 bzw. 58%)
jetzt € 7,50
Preise inkl. gesetzl. MwSt. zzgl. Service- & Versandkosten

lieferbar - bei Bestellung in den nächsten 23 Std. 33 Min.
mit 24-Stunden-Service erfolgt die Lieferung am Donnerstag.*

Anzahl: 1

MERKEN IN DEN WARENKORB
VERGLEICHEN WEITEREMPFEHLEN
JETZT BEWERTEN und 100€ gewinnen!

Grenzen der Testautomatisierung (5)

Testing Mindset

- eher orientiert auf Unvorhersehbarkeit und kritische Annäherung
- Hinterfragen, Schlussfolgern
- inkrementelle Untersuchung, um permanent
  - neues Wissen zu gewinnen
  - bekannte Grenzen zu erweitern
  - besseres Feedback zu geben

Automation Mindset

- eher technisch und algorithmisch orientiert
- sicherstellen, dass die Testfälle immer
  - in der gleichen Umgebung
  - mit den gleichen Daten
  - unter den gleichen Bedingungen
  - auf gleiche Weise ausgeführt werden

Grenzen der Testautomatisierung (6)

Es gibt keinen Ersatz für das Ausführen des Gesamtprodukts und das Testen, wie alle Komponenten zusammen funktionieren!

- ➔ Testautomatisierung ist sehr wichtig, genügt aber nicht
- ➔ Testpyramide beachten
- ➔ sinnvollen Testumfang auf jeder Ebene bestimmen
- ➔ Testautomatisierung vor allem für Regressionstests
- ➔ Neue Features zunächst umfassend manuell testen, dann Teile automatisieren
- ➔ Lücken durch andere Maßnahmen ausfüllen

Agenda

- Testautomatisierung – Grundlagen
- Testautomatisierung – Ebenen
- Grenzen der Testautomatisierung
- TDD, BDD, ATDD

Test Driven Development (1)

TDD

Idee

- Unit-Tests vor der zu testenden Komponente implementieren
- die Implementierung passt sich an die Erwartungen der Testfälle an, nicht die Testfälle an vollendete Tatsachen der Implementierung
- Entwicklung und Test verschmelzen

➔ TDD verbessert den Test und vor allem die Implementierung!

Test Driven Development (2)

TDD

Ablauf

1. Schnittstelle der zu testenden Komponente aus Black-Box-Sicht entwerfen: Klassenhierarchie, Interfaces, Methodensignaturen, ...

2. Leere Methodenkörper der Komponente compilierfähig implementieren

```
double calculateAverage( double[] numbers ) {
     return 0.0d;
}
```

3. Unit-Tests für die öffentliche Schnittstelle entwerfen

4. Unit-Tests implementieren (Normalfall, Sonderfälle, Fehlerfälle)

5. Erster Testlauf, alle Tests schlagen fehl (rot)

6. Komponente schrittweise implementieren, bis alle Tests erfolgreich (grün)

Test Driven Development (3)

TDD

Warum werden die Unit-Tests zuerst implementiert?

- Frühestmögliche Nutzung der zukünftigen Komponente durch Unit-Tests

  ➔ ungünstiger Schnittstellenentwurf schon vor Implementierung sichtbar

  ➔ da Unit-Tests möglichst isolierte Komponenten benötigen, werden ungünstige Abhängigkeiten schnell erkannt und beseitigt

  ➔ frühestmögliche Überprüfung von Code-Änderungen durch bereits vorhandene Unit-Tests

  ➔ Eigentlicher Code von Beginn an besser testbar und nutzbar!

- Tests für noch nicht implementierte Funktionen existieren, aber schlagen fehl ➔ noch zu implementierende Funktionen sind gut sichtbar

Test Driven Development (4)

TDD

Psychologische Faktoren

- Testgetriebene Entwicklung motiviert durch einen Zyklus kleiner Erfolge:
  - Durchführung Testlauf
  - fehlschlagende Tests zeigen, wo Änderungen nötig sind (rot)
  - Programmierung
  - erneuter Testlauf
  - kleiner Erfolg (grün)
- optimal sind möglichst kleine Schritte pro Zyklus

Behavior Driven Development (1)

BDD

Ausgangssituation: Unzufriedenheit mit TDD

- TDD bezieht sich auf die konkrete Implementierung (Klassen, Methoden)
- TDD hat eine rein technische, keine fachliche Sicht
- TDD nur für Entwickler verständlich
- TDD beschreibt nicht, *warum* etwas implementiert werden soll
- keine Aussagen über Anforderungen

➔ Weiterentwicklung von TDD zu BDD durch Dan North in 2006

Behavior Driven Development (2)

BDD

Idee hinter Behavior Driven Development (BDD)

- Beschreibung
  - des fachlichen Soll-Verhaltens („Behavior“), nicht der Implementierung
  - anhand von Beispielen, sogenannten Szenarien
  - unter Nutzung einer ubiquitären (allgegenwärtigen) oder domänenspezifischen Sprache
- fachliche Szenarien als „ausführbare Dokumentation“
  - treiben die Entwicklung
  - bilden auch die Testfälle (Black-Box-Ansatz)
- Trennung von
  - fachlicher Spezifikation und
  - technischer Umsetzung

➔ für Fachexperten und Kunden verständlich, sie schreiben selbst Szenarien

➔ technische Umsetzung in Testcode durch Testautomatisierer oder Entwickler

Behavior Driven Development (3)

BDD

Ablauf

- Auswahl eines BDD-Werkzeugs, Entscheidung für eine Sprache
- Format durch BDD-Werkzeuge vorgegeben, oft als „Wenn-Dann“-Sätze
- Erstellung der Szenarien
- Entwicklung der zu testenden Software; Szenarien dienen als Anforderungen
- Programmieren eines passenden Codebausteins zu jedem Fragment eines Szenarios („Fixture“) für den automatisierten Test
- Ausführung der Szenarien als Tests mit dem BDD-Testwerkzeug
  - Testwerkzeug liest die Szenarien
  - findet einen passenden Codebaustein
  - führt ihn aus
- Codebausteine sind parametrisierbar; dadurch mit unterschiedlichen Daten mehrfach nutzbar

Behavior Driven Development (4)

BDD

Definition von Szenarien laut Dan North im GWT-Stil:

```
Given some initial context (the givens),
When an event occurs,
Then ensure some outcomes.
```

Behavior Driven Development (5)

BDD

Beispiel für ein Szenario

```
Szenario: Erfolgreiches Login mit Benutzername

Gegeben die Seite „Login“ ist geöffnet
Wenn ich in das Eingabefeld „Login“ „nutzername1“ eingebe
Und ich in das Eingabefeld „Passwort“ „passwort1“ eingebe
Und ich auf den Button „Login“ drücke
Dann sollte ich auf der Seite „Kundenbereich“ sein
Und ich sollte „Vorname1 Nachname1“ sehen
```

Beispiel für ein passendes Code-Fragment im BDD-Werkzeug JBehave

```
@Given("die Seite $page ist geöffnet")
public void aPage(String page) {
   // ...
}
```

Behavior Driven Development (6)

BDD

BDD ist hilfreich, kann aber nicht alle Ziele einlösen

- Szenarien müssen doch einer formalen Sprache folgen
  - ➔ die angestrebte Ausdrucksfähigkeit einer natürlichen Sprache geht verloren
  - ➔ neue Schlüsselwörter/Phrasen erfordern Entwicklung neuer Fixtures
- Komplexität durch zwei voneinander abhängige Ebenen (Szenarien + Fixtures)
  - ➔ schwer zu verwalten, schwer zu warten
  - ➔ Welche Schlüsselwörter/Phrasen sind bereits definiert?
  - ➔ schnell unübersichtlich
- Wahl der Abstraktionshöhe in den Szenarien schwierig
  - Schlüsselworte für einzelne Bedienaktionen ➔ Szenarien unübersichtlich
  - Schlüsselworten für komplexere Funktionen ➔ Fixtures komplexer
- zusätzliche Sprache, zusätzliche Werkzeuge erforderlich
- Kunden und Product Owner schreiben in Praxis trotz BDD meist keine Szenarien

Acceptance Test Driven Development (1)

ATDD

Idee und Ablauf

- Lücke schließen, gemeinsames Verständnis zwischen
  - fachlichen Anforderungen aus Kundensicht und
  - technischem Ablauf aus Entwickler- und Testersicht
- Erstellung von Anforderungen mit Akzeptanzkriterien an beschreibenden Beispielen:
  - gemeinsam durch Kunden, Product Owner, Entwickler, Tester
  - in einer gemeinsamen, verständlichen Sprache
  - daraus Erstellung automatisierter Akzeptanztests vor der Implementierung
  - diese treiben die Entwicklung ➔ *„Acceptance Test Driven Development (ATDD)“*
  - hohe Anforderungsabdeckung von fast 100% anzustreben
- sehr ähnlich BDD, oft BDD-Werkzeuge auf gleiche Art genutzt
- auch als *Storytest Driven Development (STDD)* bezeichnet
- Werkzeuge: Cucumber, Jnario, NatSpec, ...

Acceptance Test Driven Development (2)

ATDD

Kritik

- oft kontrovers diskutiert, oft unterschiedliches Verständnis
- Praxistauglichkeit fraglich
- Kunden und Product Owner
  - denken oft nicht in Akzeptanzkriterien
  - möchten oder können Akzeptanzkriterien nicht erstellen
- testet eher auf Integrations- und GUI-Ebene
- dreht die Testpyramide eher um; fokussiert auf die Spitze der Pyramide
- fördert große Upfront-Designs, was Agile Vorgehensmodelle vermeiden möchten
- eher zeitaufwendig statt leichtgewichtig
- Tests müssen nach Implementierung oft stark geändert werden

➔ aus unserer Sicht ebenfalls schwer umsetzbar, fällt oft auf BDD zurück

Vielen Dank für Ihre Aufmerksamkeit!

Haben Sie Fragen?

Kontakt

Xceptance Software Technologies GmbH
Leutragraben 2-4
07743 Jena

Tel.: +49 (0) 3641 55944-0
Fax: +49 (0) 3641 376122

E-Mail: kontakt@xceptance.de
http://www.xceptance.de

Copyrights