Die Techniker

Unit Tests sind blöd!

Grenzen und Chancen

Sven Grand

Juni 2018

Inhalt
Unit Tests sind blöd!

1. Testautomation und Unit Tests
2. Eine Unit Test Definition
3. Unit Tests sind blöd
   - nur blöd
   - richtig blöd
4. Unit Tests machen Sinn
5. Testautomationsstrategie
6. Referenzen

Unit Tests sind blöd aber in einer Testautomationsstrategie unverzichtbar.

Testautomation und Unit Tests

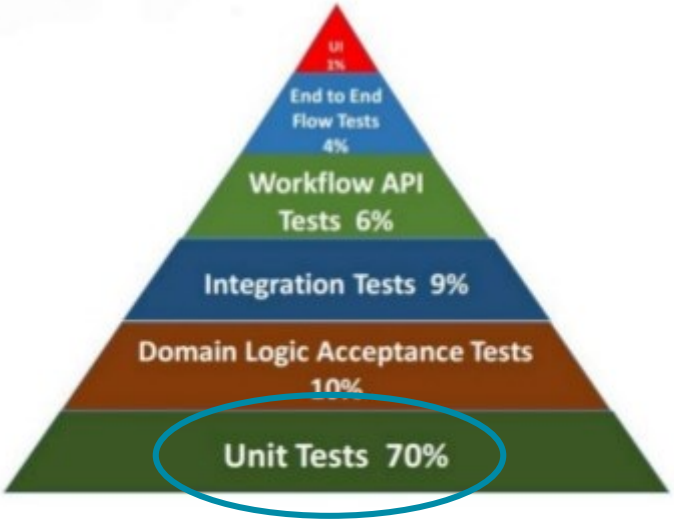
Wandel in der Softwareindustrie

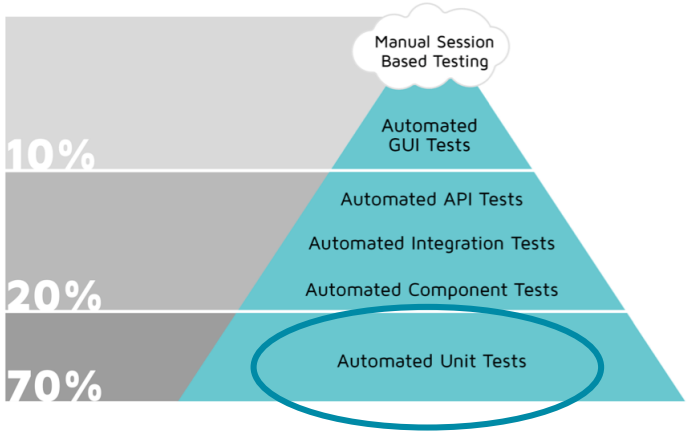
- Kürzere Releasezyklen
  - Amazon: alle 11.6 sec
  - Flickr: 10+ pro Tag

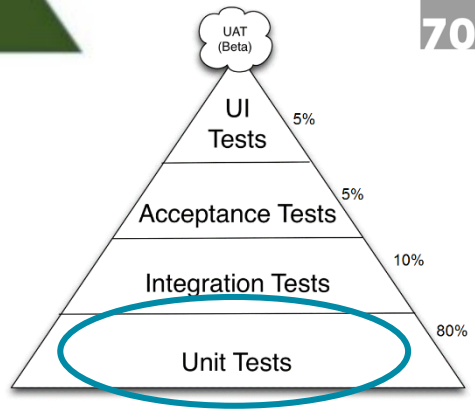
- Hohe Qualität **und** kurze Releasezyklen
- Nur mit Testautomation zu schaffen

Testautomation und Unit Tests

Drei Beispiele







Eine Unit Test Definition

- Testet eine Klasse **isoliert** von anderen Klassen
- Läuft im Hauptspeicher **ohne**
  - Zugriff auf das **Dateisystem** oder **Datenbanken**
  - Zugriff auf das **Betriebssystem**
  - Zugriff auf das **Netzwerk**
  - Zugriff auf **Fremdsoftware**
- Testet keine **Threads**

Eine Unit Test Definition

- **Value Objects, Strings, Lists, Collections** usw. werden mitgetestet
- Unit Tests erzeugen keine **zufälligen** und keine **großen** Testdaten
- Testet nur den von **uns** geschriebenen Source Code
- Unit Tests werden **vom Entwickler** in **derselben** Programmiersprache geschrieben
- Jede Funktionalität, die wir mit Unit Tests testen können, nennen wir **Basisfunktionalität**

Der Chor der Kritiker

Code an Tests anpassen? Geht's noch?

Dependency Injection? Wozu?

Zu viele Interfaces überblickt keiner mehr.

Meine Komponente ist speziell. Man kann sie nicht Unit-testen.

Unit Tests finden keine Fehler. Probleme liegen doch im komplexen Zusammenspiel.

Unit Tests sind blöd

Eigene Klassen voneinander isolieren

Problem	Aktion
Abhängigkeiten leicht austauschbar machen	• Interfaces für eigene Klassen • Dependency Injection/Service Locator

Zu viele Abhängigkeiten/ ‚Train Wrecks‘
 	
Beispiel:
 	
`service.getMortgage()`
`.paymentCollection()`
`.getNextPayment()`
`.applyPayment(300)`
	Mock-freundliche Schnittstellen/ ‚Law of Demeter‘
	
	Beispiel:
	
	`service.applyMortgagePayment(300)`

Unit Tests sind blöd

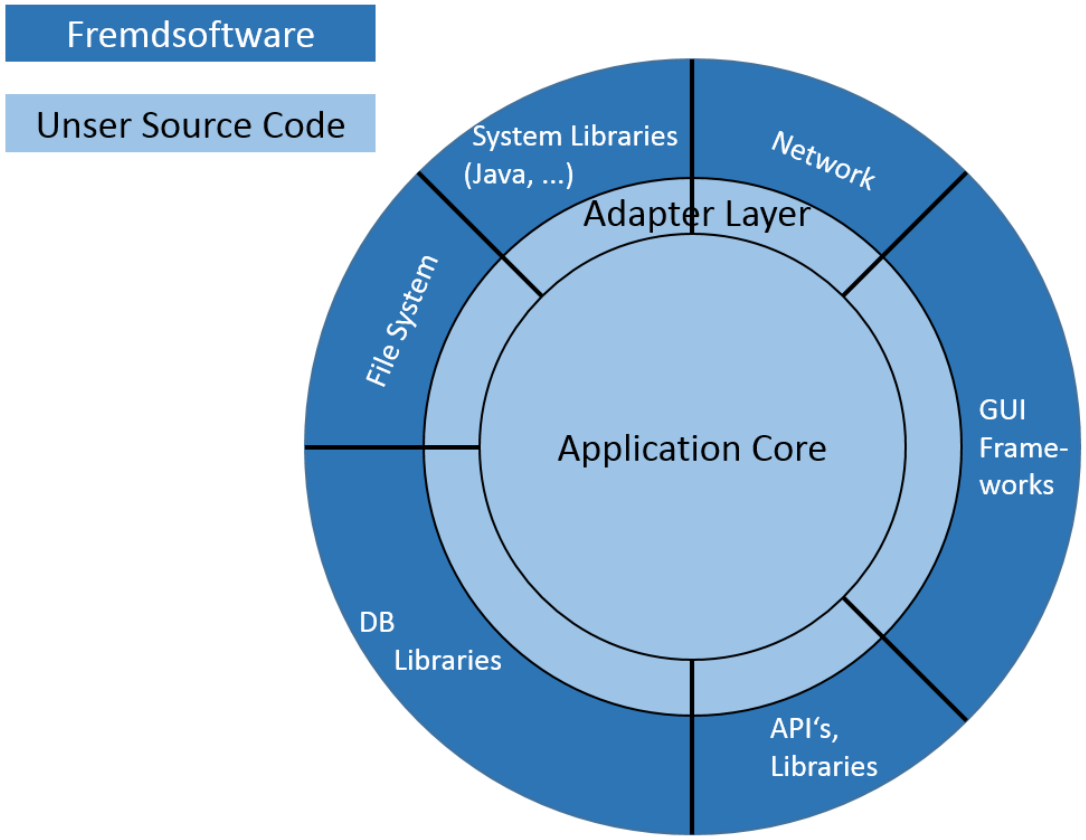
Eigene Klassen von Bibliotheken isolieren

Problem	Aktion
Bibliotheksaufrufe im Test austauschbar machen	Kapselung durch Adapter
Bibliotheksaufrufe einfacher ‚mock-bar' machen	Vereinfachung durch Adapter

Beispiel: Adapter zur Datenbank

```
public interface IVersichertenRepository {
      public Versicherter load(String id);
      public void save(Versicherter vers);
}
```

Unit Tests sind blöd
Wir brauchen Adapter



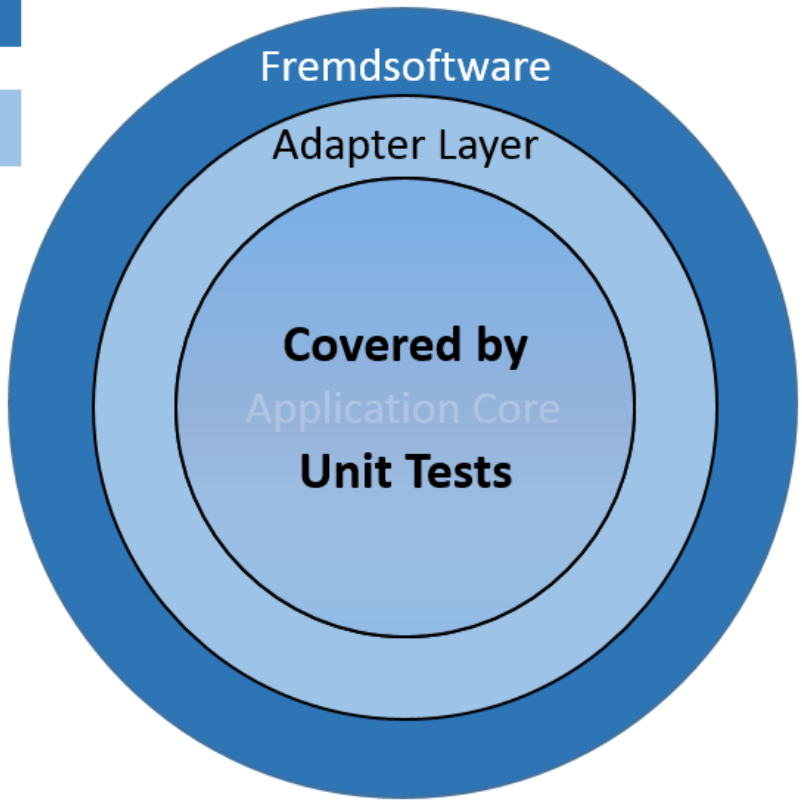
- Vgl.: A. Cockburn, E. Evans, S. Freeman & N. Pryce, R. Martin

Unit Tests sind blöd

Sie können keine Adapter testen

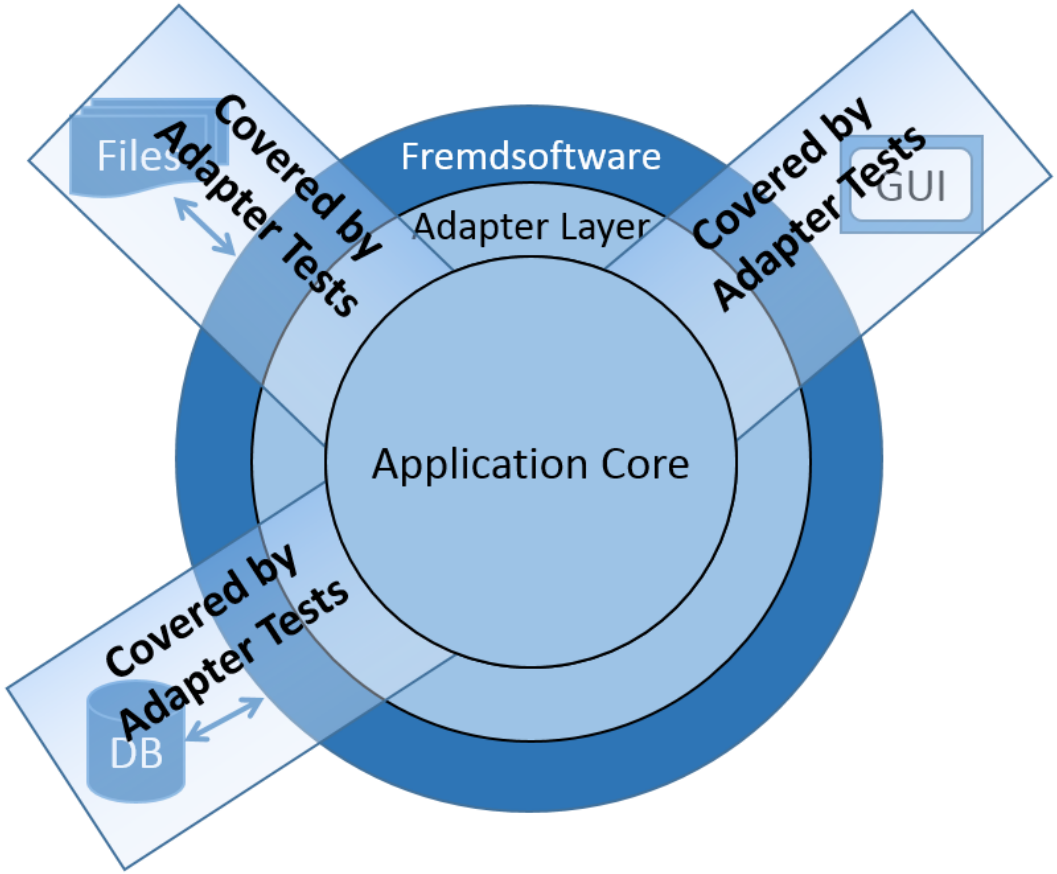
Fremdoftware

Unser Source Code

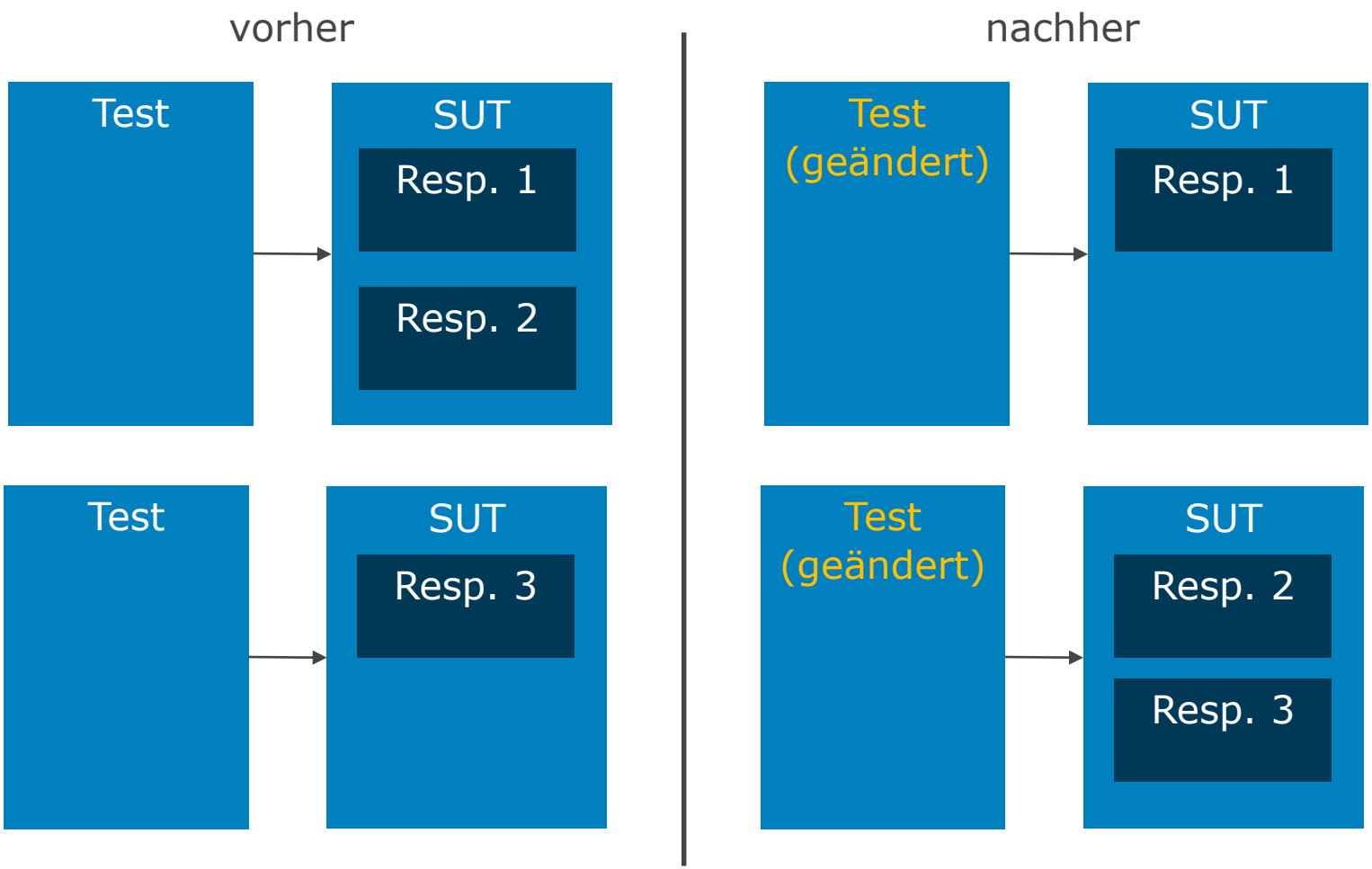


Unit Tests sind blöd

Wir brauchen zusätzlich Adapter-Tests

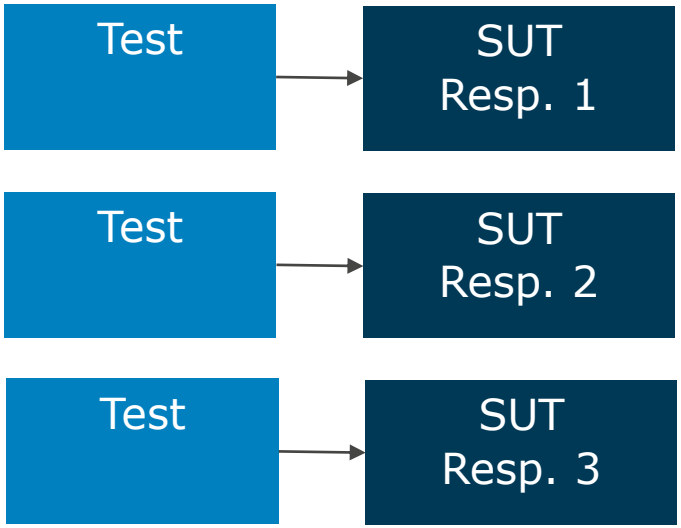


Unit Tests sind blöd
Wartung von Tests



Unit Tests sind blöd
Wartung von Tests

- Single Responsibility Principle



- Produktionscode einfacher
- Tests einfacher

Unit Tests sind blöd
Sie finden keine Fehler

- Vom Entwickler selbst geschrieben
- Vom Entwickler erwartetes Verhalten
- Nach Implementierung ‚grün‘

➢ gefühlt werden keine Fehler gefunden

- Refactorings, Erweiterungen, Änderungen
- Entwickler hat Produktionscode einen Monat nicht angeschaut
- Entwickler hat den Produktionscode nicht selber geschrieben

➢ Fehler können gefunden werden

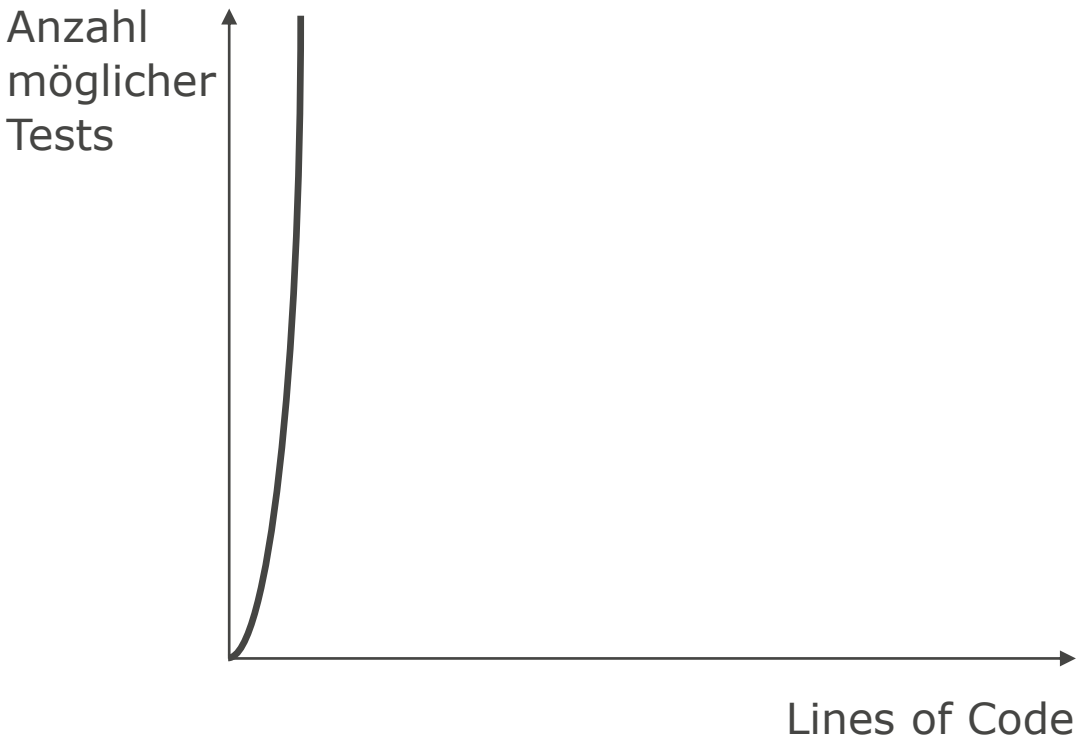
Unit Tests sind richtig blöd,
wenn sie schlecht geschrieben sind

Beispiele für gängige Probleme sind:

- Keine Unit Tests
  - zu viele Voraussetzungen
  - zu langsam
  - zu fragil
- Zu viele Mocks statt Stubs
- Mocks/Stubs für fremden Code
- Zu viel „Setup" Code
- Wo ist der „Execute" Teil?
- Zu viel „Verify" Code
- Gar kein „Verify" Code

Unit Tests sind richtig blöd

Keine formale Korrektheit



Unit Tests sind richtig blöd

Unit Tests können nicht sicher stellen,

- dass alle Teile unseres Codes richtig miteinander verdrahtet sind
- dass unsere Klassen gegenseitig ihre Schnittstellen korrekt benutzen
- dass die Fremdsoftware funktioniert
- dass unsere Klassen die Fremdsoftware korrekt benutzen
- dass die Funktionalität aus Sicht der Benutzer vollständig und korrekt ist

Unit Tests sind richtig blöd

Unit Tests können nicht sicher stellen,

- E2E Tests
- Integrated Tests, Design by Contract
- Adapter Tests
- Adapter Tests
- Acceptance Tests

Unit Tests sind richtig blöd

Unit Tests können nicht sicher stellen,

- dass wir keine Nebenläufigkeitsprobleme haben
- dass unser Code ausreichend schnell und speichersparend ist
- dass unser Code ausreichend Last bewältigen kann
- dass unser Code ausreichend lange laufen kann

- dass das UI richtig mit der Funktionalität verdrahtet ist

- dass das UI ergonomisch ist
- dass unsere Software das tut, was die Benutzer brauchen

Unit Tests sind richtig blöd

Unit Tests können nicht sicher stellen,

- Performance Tests,
- Stress Tests,
- Duration Tests
- GUI Adapter Tests, GUI-E2E Tests
- Manuelle Usability Tests,
- Manuelle Exploration Tests

Unit Tests machen Sinn

- Sehr schnelles Feedback
- Sind häufig und auf jedem Rechner ausführbar
- Keine speziellen Rechner müssen konfiguriert und betrieben werden
- Keine Zeitverschwendung für Analysieren und Fixen von Tests
- Tests auf einen Blick verständlich
- Komplexität moderat
- Fehlersituation lassen sich sehr einfach simulieren
- Robust und zuverlässig
- Machen die Architektur des Produktions-Codes flexibler

Unit Tests machen Sinn

"A majority of the production failures (77%) can be reproduced by a unit test"

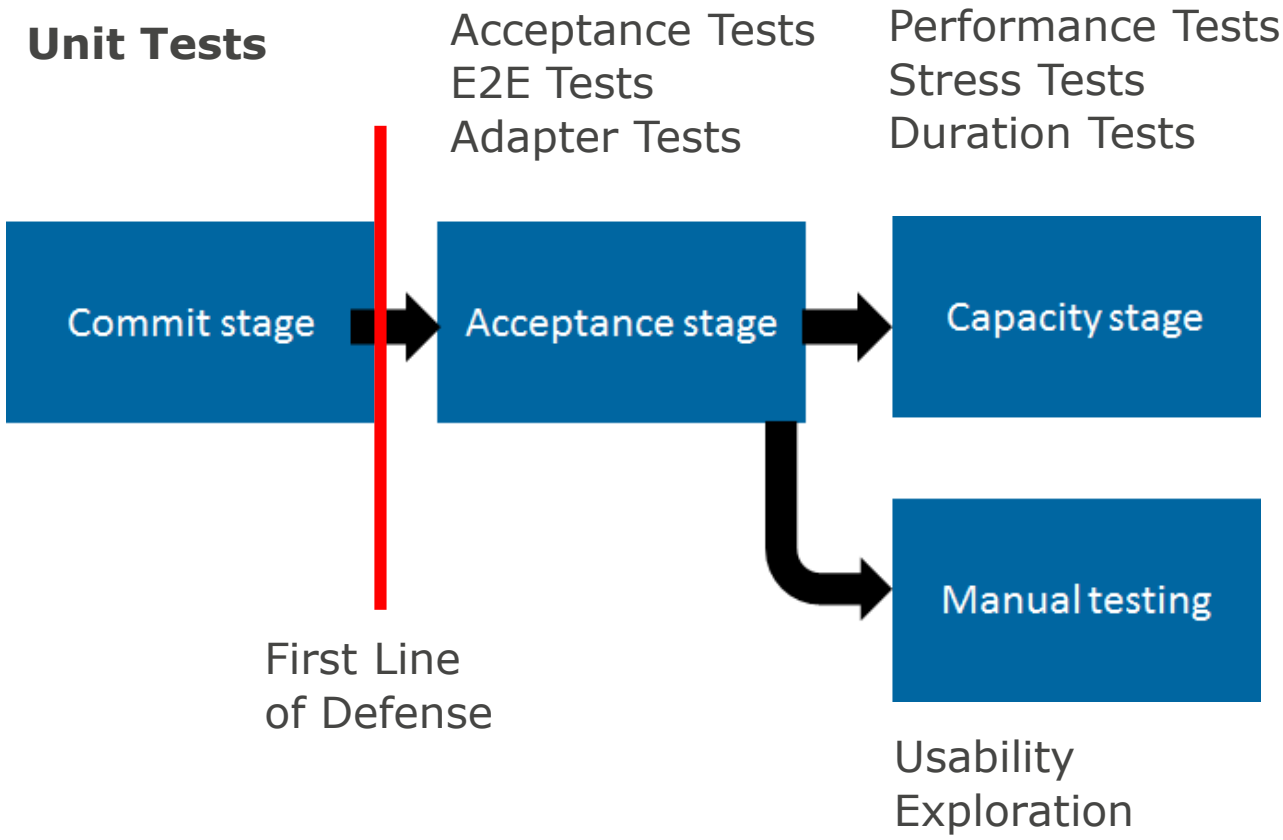
Simple Testing Can Prevent Most Critical Failures: An Analysis of Production Failures in Distributed Data-Intensive Systems

Ding Yuan, Yu Luo, Xin Zhuang, Guilherme Renna Rodrigues, Xu Zhao, Yongle Zhang, Pranay U. Jain, and Michael Stumm, *University of Toronto*

11th USENIX Symposium on
Operating Systems Design and Implementation.
October 6–8, 2014 • Broomfield, CO

Testautomationsstrategie

Hohe Qualität und kurze Releasezyklen



Vgl.: J. Humble, D. Farley: Continuous Delivery

Fragen?

Sven Grand

Techniker Krankenkasse
sven.grand@tk.de

Referenzen

- J. Jenkins: „Amazon deploys new software to production every 11.6 seconds”
  http://joshuaseiden.com/blog/2013/12/amazon-deploys-to-production-every-11-6-seconds/
- J. Allspaw: „10+ Deploys Per Day at Flickr”
  https://de.slideshare.net/jallspaw/10-deploys-per-day-dev-and-ops-cooperation-at-flickr/77
- Wie kann man Threads aus Unit Tests heraushalten?
  - siehe z.B.: http://jmock.org/threading-executor.html
  - siehe z.B.: S. Grand, Async Programming : Unit Testing Asynchronous Code: Three Solutions for Better Tests, MSDN Magazine Nov 2014, https://msdn.microsoft.com/en-gb/magazine/dn818494.aspx

Referenzen

- W. Cunningham, http://wiki.c2.com/?TrainWreck
- W. Cunningham, http://wiki.c2.com/?LawOfDemeter
- “Ports and Adapters Architecture”, A. Cockburn, http://alistair.cockburn.us/Hexagonal+architecture
- “Anticorruption Layer”, E. Evans, Domain-driven Design: Tackling Complexity in the Heart of Software, Addison-Wesley, 2003
- “Interface Adapters”, R. Martin, Clean Architecture, Prentice Hall, 2017
- “Adapter Layer”: S. Freeman, N. Pryce, Growing Object-Oriented Software, Guided by Tests, Addison-Wesley, 2009

Referenzen

- https://en.wikipedia.org/wiki/Single_responsibility_principle
- Begriffsklärung für alle Arten von “Test Doubles”:
  G. Meszaros , xUnit Test Patterns: Refactoring Test Code, Addision Wesley, 2007
- Simple Testing Can Prevent Most Critical Failures: An Analysis of Production Failures in Distributed Data-Intensive Systems”, https://www.usenix.org/system/files/conference/osdi14/osdi14-paper-yuan.pdf
- J. Humble, D. Farley, Continuous Delivery: Reliable Software Releases Through Build, Test, and Deployment Automation, Addision Wesley, 2011