Putting TDD to the Test

Edwin Günthner, IBM Germany Development Lab

Zum Titel

www.dict.cc:

to put sb./sth. to the test

jdn./etw. auf den Prüfstand stellen

Agenda

- Teil 1: Grundsätzliches, Begrifflichkeiten, Definitionen
- Teil 2: Ein (subjektiver) Reisebericht
- Teil 3: Eine (objektivere) Abrundung

Teil 1

- Motivation
- Definitionen

Warum sind wir heute hier?

*Continuous attention to technical excellence and good design **enhances** agility.*

[Agile manifesto, 9th principle]

Definitionen: Unit Testing

- Unit Testing [Wikipedia]:

  Vereinfacht gesagt steht dort:

  „Alles was man tut um zu testen … ist ein Unit Test“

→ Nicht hilfreich

Definitionen: Unit Testing

- Hilfreich:
  - https://de.wikipedia.org/wiki/Modultest
  - http://artofunittesting.com/definition-of-a-unit-test/

  → Kennzeichen *guter* Unit Tests sind zum Beispiel:
    - Automatisierbarkeit
    - Isolation
    - Konsistenz
    - Schnelligkeit

Definitionen: TDD

- TDD: Test driven development [Wikipedia]:

  *Test-driven development (TDD) is a software development process that relies on the **repetition of a very short development cycle**: requirements are turned into very specific test cases, then the software is improved to pass the new tests, only.*

→ Uns geht es hier um **TDD** und ***gute, echte*** Unit Tests

Teil 2

- Evolution und Kontext
- *Houston, we have a problem*
- *TDD to the rescue*
- Erfahrungen und Ergebnisse

Evolution I

- 2012
  Kollegen (und Java Forum Stuttgart) … alles spricht über TDD

  Aber: dem Team einfach sagen:
  „Wir machen jetzt alle TDD“ … funktioniert nicht (besonders gut)

Evolution II

- Wir üben, üben, üben …
- 2015 ergibt sich für mich die Chance, eine komplexe Funktionalität komplett mit TDD zu bearbeiten

Kontext

- Das ist ein Mainframe:

- … ein großes SMP (symmetric multi-processing) System, auf dem Kunden *virtuelle Maschinen* betreiben

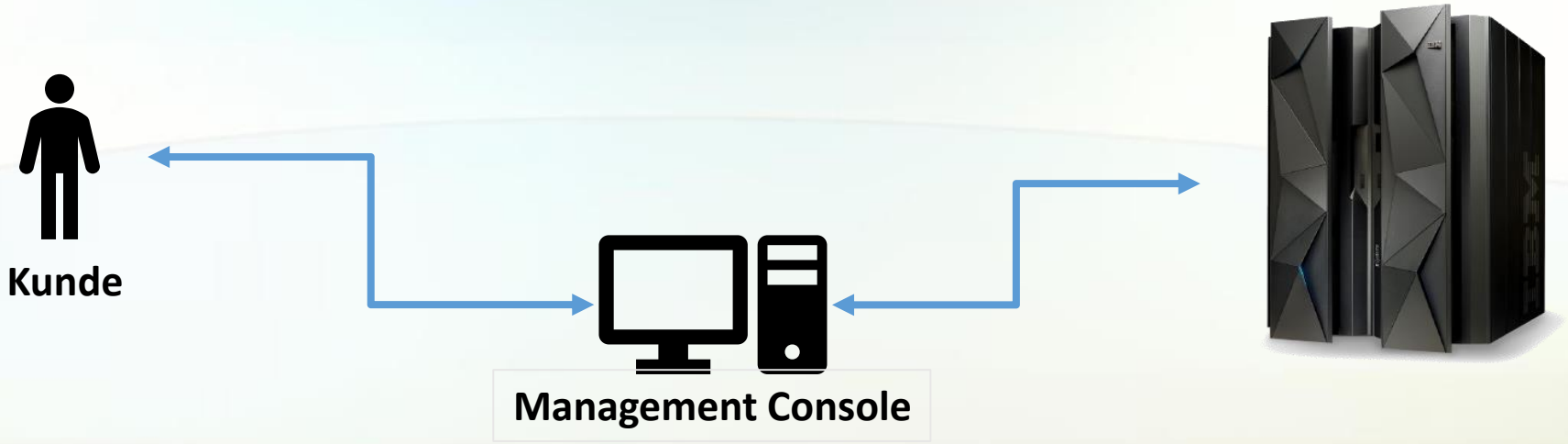
Kontext

- Das ist ein Mainframe:

- … ein großes SMP (symmetric multi-processing) System, auf dem Kunden *virtuelle Maschinen* betreiben

- Die IBM zSystems Komponente DPM ermöglicht Kunden die *zeitgemäße* Verwaltung dieser VMs

Kontext (II)



- *Management Console*:
  - (physikalisch) verteilte Anwendung
  - xx Millionen Zeilen C, C++, Java
  - *historisch gewachsen*, proprietär

Houston, we have a problem

- Ein wesentlicher Teil von DPM: das Starten/Stoppen der VMs
- Erster Code entsteht Frühjahr/Sommer 2015
- Code Review Herbst 2015:
  - Code ist bereits relativ komplex – aber wichtige Funktionen fehlen noch
  - Viel *code duplication* – aber keine Unit Tests

- Das *eigentliche* Problem: Refactoring ist ohne Unit Tests **zu teuer**

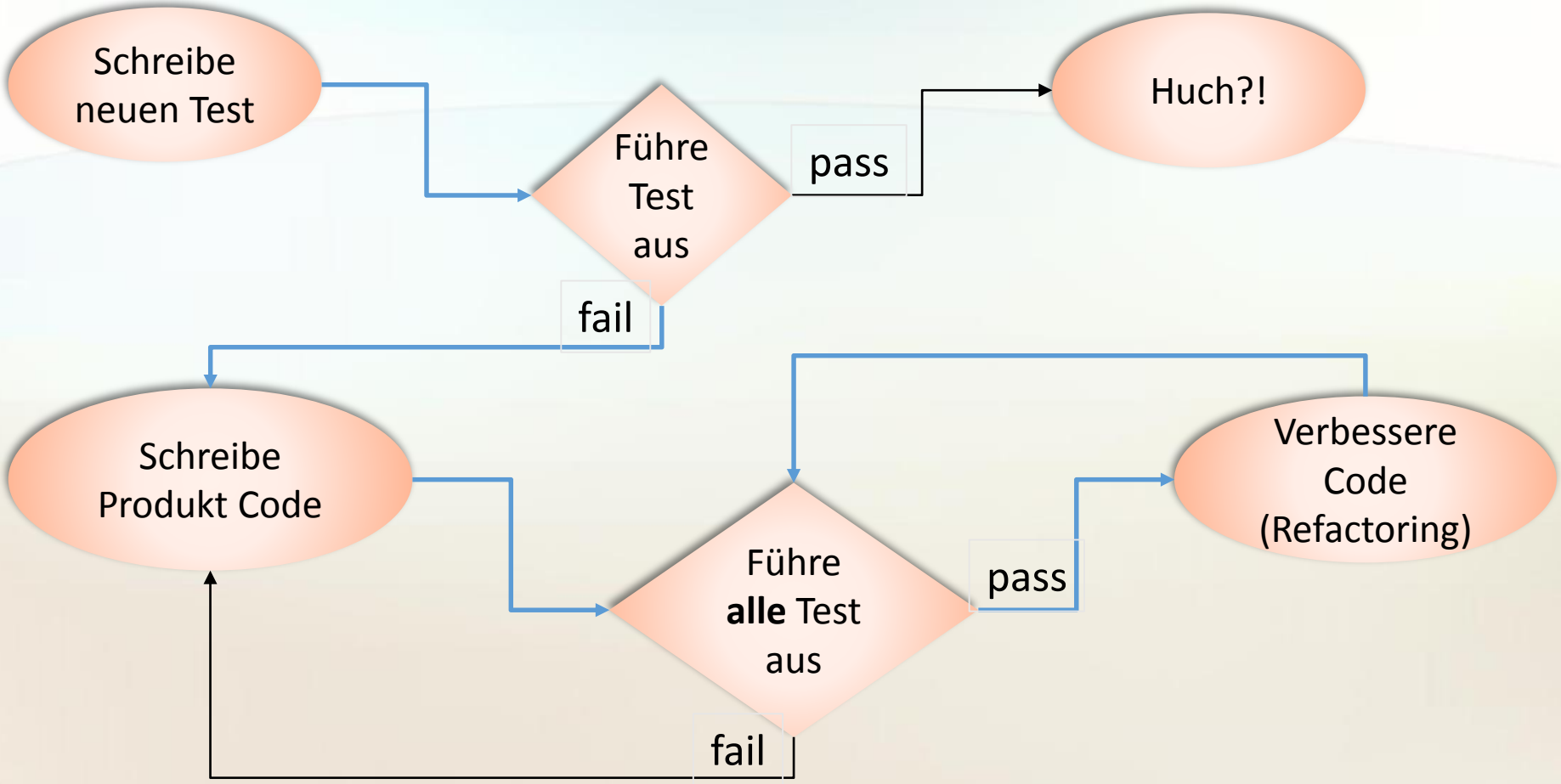
Die Lösung: *the big rewrite*

- Anfang Dezember 2015:

  Kein Urlaub mehr 2015 … also nutze ich die ruhigen Wochen zum Jahresende und überarbeite alles

  *Und weil ich es besser machen will, mache ich **alles** mit TDD*

Wie geht das nochmal mit TDD?



Die erste und wichtigste Erfahrung

- Die kurzen *Test Code* <--> *Produkt Code* Zyklen sind ***paradiesisch***:
  - Ich arbeite *ausschließlich* in meiner IDE
    (versus: Code schreiben, Patch zusammenstellen, VM starten,
    Patch applizieren, Anwendung starten, …)
  - Ich kann eine Idee/Vorstellung ***sofort*** ausprobieren und umsetzen
- Mit anderen Worten:
  - Kein Wechseln des Kontexts
  - Keine Wartezeiten
  - Im Gegenteil: schnelle *rot* → *grün* Iterationen … stunden-, tagelang

Die zweite Erfahrung

- Nach 4 Wochen: ein erster **funktionaler** Test am echten System „Huch, es tut *nicht*!“

Die zweite Erfahrung

- Nach 4 Wochen: ein erster **funktionaler** Test am echten System „Huch, es tut *nicht*!“
- Analyse ergibt:
  - Fehler beim Erstellen des *Patches*
  - Neuer Patch: Test läuft erfolgreich durch
  - Aber: da ist ein Fehler im *Design* der neuen Funktionalität

Die zweite Erfahrung

- Nach 4 Wochen: ein erster **funktionaler** Test am echten System
  „Huch, es tut *nicht*!“
- Analyse ergibt:
  - Fehler beim Erstellen des *Patches*
  - Neuer Patch: Test läuft erfolgreich durch
  - Aber: da ist ein Fehler im *Design* der neuen Funktionalität
- Dank TDD und existierender Unit Tests:
  Fehler ist in wenigen Stunden behoben

All good things must come to an end …

- Nochmal etliche Wochen später: *coding complete*
- Zeit für echte *System Tests*:
  - Alle Tests laufen erfolgreich durch

All good things must come to an end …

- Nochmal etliche Wochen später: *coding complete*.
- Zeit für echte *System Tests*:
  - Alle Tests laufen erfolgreich durch
- Genauer gesagt:
  - Keine *Übertragungsfehler*
    (alles was vorher funktioniert hat, funktioniert immer noch)
  - Alle bekannten Probleme im alten Code sind ebenfalls beseitigt
  - Große Mengen an nützlicher Zusatzfunktionalität wurden eingebaut

und **alles** funktioniert auf Anhieb

In der Retrospektive

- Der neue Code wird seit 18 Monaten genutzt:
  keine Bugs gefunden

- Der neue Code wurde mehrmals erweitert
  (z. B. für andere Operationen wie *Partition Update*) – ohne dass dabei Regressionen eingebaut wurden

  (Wichtig: existierende Unit Tests helfen nur bedingt beim Implementieren neuer Funktionen)

Teil 3

- Alles gut?
- Ein Geheimtipp
- Effizienz
- Best Practices
- Empfehlungen und Anti-Pattern

Alles gut? Natürlich nicht.

- Aufwandsabschätzung
  (überraschenderweise: viel zu niedrig)
- Keine konsequente Umsetzung der eigenen Qualitätsansprüche
  (zum Beispiel: „Keine Zeit für Reviews“)
- Funktionale Tests sind immer noch ein **MUSS**

→ TDD ist kein *silver bullet* mit dem Anspruch, *alle* Probleme zu lösen!

Talking about silver bullets

- Unit Tests funktionieren für …

? ? ?

Talking about silver bullets

- Unit Tests funktionieren für **testbare Units!**

Was macht eine „Unit“ testbar?

- Zum Beispiel:
  - Isolation
  - Entkopplung
  - „Fokussierung“ (Single Responsibility Principle)

→ die Prinzipien von **Clean Code**!

Die ärgerliche Konsequenz

- Es genügt nicht, Bücher über Unit Testing / TDD zu lesen, z. B.:
  - „JUnit-Profiwissen“ [Michael Tamm]
  - „xUnit Test patterns“ [Gerad Meszaros]
  - „Working Effectively with Unit Tests“ [Jay Fields]

Die ärgerliche Konsequenz

- Es genügt nicht, Bücher über Unit Testing / TDD zu lesen, z. B.:
  - „JUnit-Profiwissen“ [Michael Tamm]
  - „xUnit Test patterns“ [Gerad Meszaros]
  - „Working Effectively with Unit Tests“ [Jay Fields]
- Darüber hinaus **muss** man sich mit *Code Qualität* **intensiv** auseinandersetzen, z. B:
  - „Refactoring“ [Kent Beck / Martin Fowler]
  - „Clean Code“, „Agile principles“ [Robert Martin, …]

So viel Aufwand? Und das rentiert sich?

- Ein Kollege hat mich gefragt:
  „Wenn man all diese Tests schreiben muss,
  wird man dann nicht **langsamer**?“
- Meine Antwort: „Nein. Man wird **schneller**.“

So viel Aufwand? Und das rentiert sich?

- Ein Kollege hat mich gefragt:
  „Wenn man all diese Tests schreiben muss,
  wird man dann nicht **langsamer**?
- Meine Antwort: „Nein. Man wird **schneller**.“

- Wieso bin ich schneller wenn ich

  n Zeilen *Produkt Code*
  \+ ***m*** Zeilen *Test Code*

  schreiben muss … anstatt nur ***n*** Zeilen?

Eine Frage der Effizienz

- Wieso bin ich schneller wenn ich ***n*** + ***m*** Code schreiben muss … anstatt nur ***n*** Zeilen?

- Ganz einfach: weil ich in wenigen Sekunden 10, 50, 100 Tests ausführen kann!

  → Unit Tests geben sofortiges Feedback über **ausgeführten** Code

Nützliche Tipps (Produkt Code)

- *Unveränderliche* Objekte machen das Leben **viel** einfacher!
- Sparsamer Einsatz *von* ***new*** und **static**
  → *dependency injection*, siehe Google Tech Talks: Unit testing
- Üben, üben, üben:
  - am Besten im Team
  - Gemeinsame Code Reviews mit beispielhaften Schwerpunkten:
    - Clean Code Prinzipien
    - Testbarkeit

Nützliche Tipps (Test Code)

- Der *ideale* Test kommt **ohne** *Mocking Framework* aus

- Für *Test Code* gelten die **gleichen** Ansprüche an Qualität wie für *Produkt Code*
  - Gemeinsame Standards und Vorgehensweisen im Team
  - Clean Code (mit gewissen Einschränkungen)

Anti-Pattern für Tests

- *„Wir brauchen PowerMock(ito)“*:
  → Bedeutet (fast) immer:
  „Wir schreiben schlecht testbaren Code“
- Tests bestehen (fast) nur aus *Mocking Specs*:
  → wir wollen das *WAS* testen, nicht das *WIE*
  → stattdessen: Fokussierung auf den ***public contract***
- Tests *fühlen* sich zu *groß/komplex/unhandlich* an:
  → vermutlich ist der *Produkt Code* zu kompliziert
  → Zeit für echtes Refactoring!

Evolution III

- … 2017

  Die Lernkurve geht *immer* weiter, mit Fragen wie zum Beispiel:

  Sollte man Unit Tests auch mal löschen?

  Wie kann ich die Nützlichkeit eines Tests sinnvoll bewerten? (Stichwort: Return on Investment!)

Fragen?

- Antworten!

  Ansonsten einfach später an:

  edwin.guenthner ät de.ibm.com

- Vielen Dank für Ihre Aufmerksamkeit!

© Copyright IBM Corporation 2017. All rights reserved.