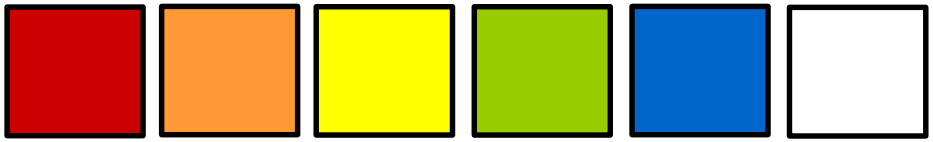


Agiles Testen

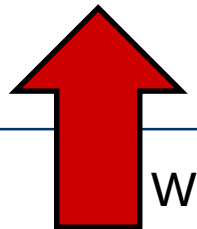
Lutz Prechelt
Institut für Informatik, Freie Universität Berlin

Wandel der Natur des Testens (brutal vergröbert)

- Grobkörnige SW-Entwicklg:
  - 1-2 Freigaben/Jahr
  - Codeänderungen vermeiden
  - Entwurf im Voraus
- → Traditionelles Testen:
  - wenig Testautomatisrng
  - praktisch kaum Modultests
  - meiste Tests über UI
  - Testpläne als Dokument
    - → oft ungleich der Realität
  - Manuelle Testprotokolle, meistens gar keine

- Continuous Deployment:
  - 100-1000 Freigaben/Jahr
  - ständige Änd. + Refactoring
  - Emergenter Entwurf
- → Agiles Testen:
  - viel Testautomatisierung
  - feinkörnige Modultests
  - Tests über UI ~vermeiden
  - Lesbare Testskripte
    - → Spezifikation per Beispiel
  - Weitreichende digitale Testprotokolle

Wahl abhängig von der Art des SW-Produkts!



Wir

Ziele von Testen
(wieder brutal vergröbert gesagt)

Freie Universität Berlin

- Traditionelles Testen:
  - Korrektheit/Zuverlässigkeit
    - +diverse nichtfunktionale Eigenschaften
- Weg dorthin:
  - Alle Register ziehen! z.B.:
  - Viel Zeit für das Testen nehmen
  - Spezialisierte Tester
  - Viel exploratives Testen
- (In Wirklichkeit gibt es verschiedene Schulen)
  - Quelle: Pettichord auf Basis von Kaner, Bach, Marick

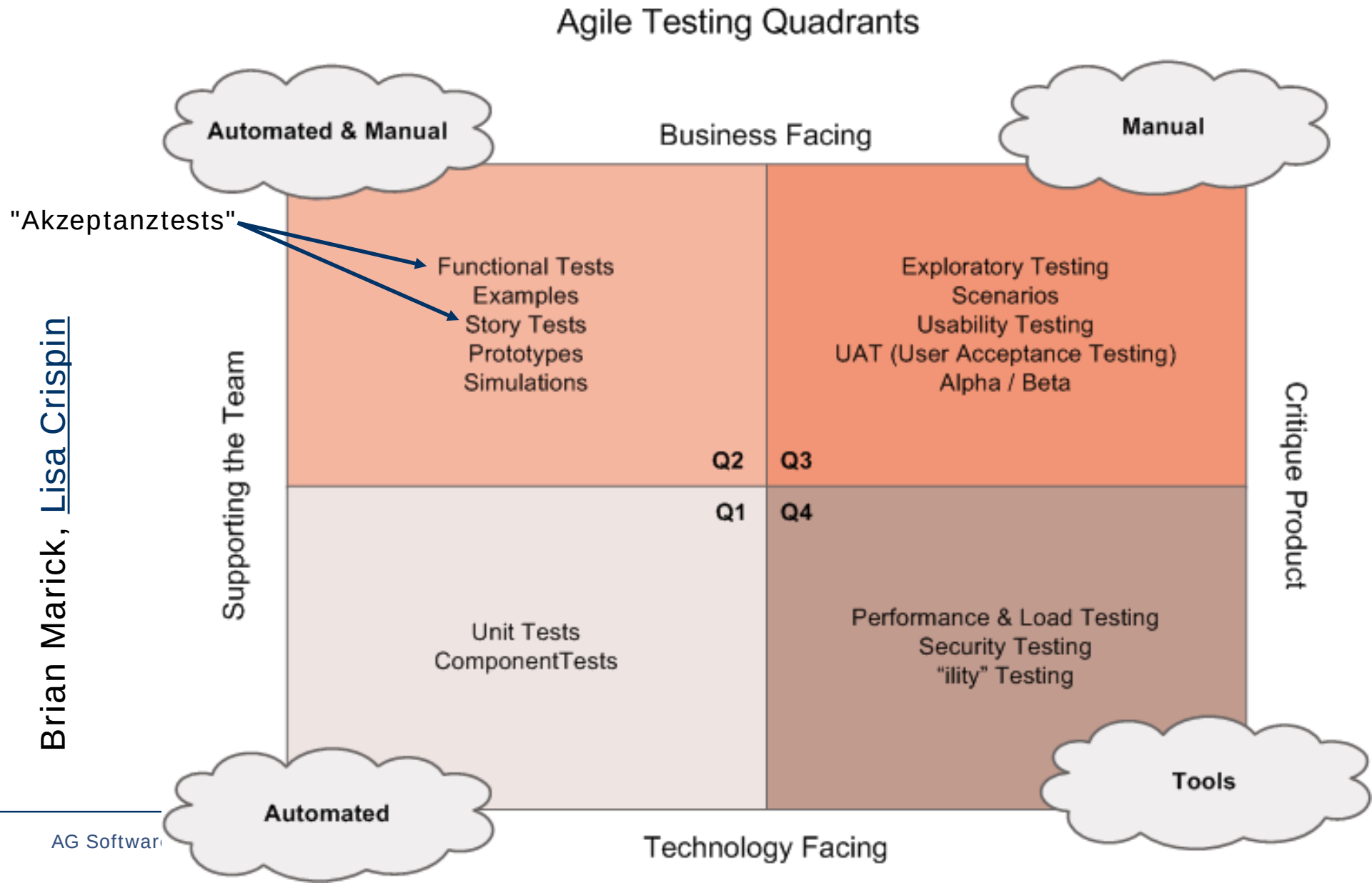
- Agiles Testen:
  - Evolvierbarkeit
  - Korrektheit (+diverse …)
- Weg dorthin:
  - **Selbsttestender Code**
  - mit wenig Testredundanz
    - eine Form von DRY
  - und hoher Test-Ablaufgeschwindigkeit
    - weil das Debugging mühsam wird, wenn man die Tests erst nach vielen Änderungen ausführt
  - Videozitat Fowler
    - 1;24:19-24:57

Wie bekommt man selbsttestenden Code hin?

- Wie bekommt man
  - und zwar mit erträglichem Aufwand
  - selbsttestenden Code
  - mit wenig Testredundanz
  - und hoher Testgeschwindigkeit?
- Und zwar
  - sowohl für Verifikation
    - Testen gegen Spezifikation
  - als auch für Validierung
    - Testen gegen tatsächliche Anforderungen?

- Elemente (E1…E4):
  1. Testarten unterscheiden
  2. Jeweils gut passende Werkzeugunterstützung bereit stellen
  3. Methoden in Erwägung ziehen:
     1. Attrappen, Testisolation
     2. Teststile
     3. Testgetriebene Entwicklung
  4. Immer schön Maß halten!
     - Bei Anforderungen an Testabdeckung
     - Stilanforderungen
     - Isolation
     - Geschwindigkeitsanford.
     - TDD-Einsatz

Element 1 (E1):
The Agile Testing Quadrants

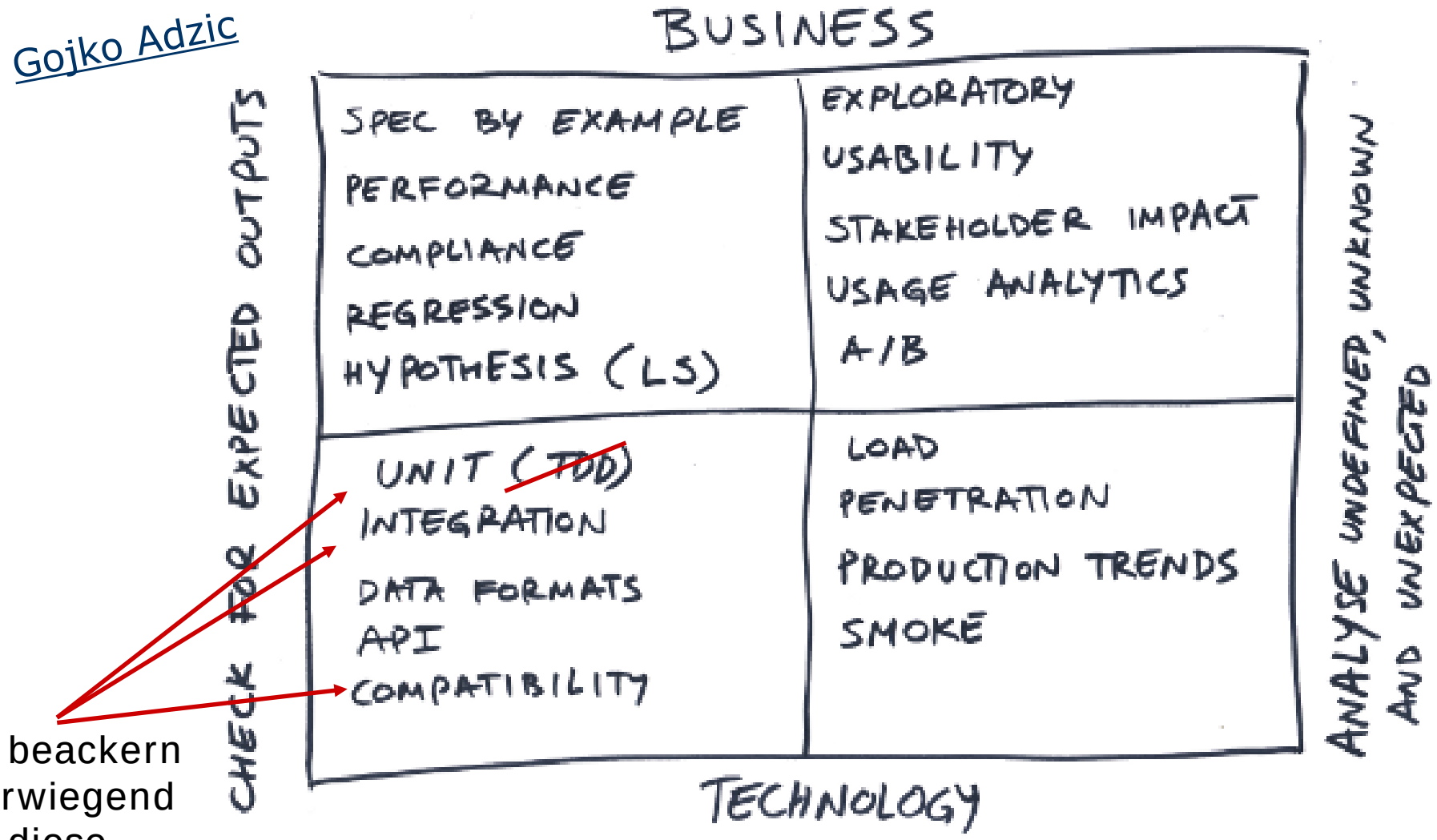


Brian Marick, Lisa Crispin

Element 1 (E1):
Agile Testing Quadrants, modernisiert

Freie Universität Berlin

- horizontal: "check" vs. "analyze" anstatt "Team" vs. "Product"



(E1,E2) Technische Tests (Techies): Unittests (Modultests)

Freie Universität Berlin

- Ziel/Zweck:
  - Korrektes Funktionieren eines einzelnen Moduls gründlich prüfen
- Merkmale:
  - stets automatisiert
  - feinkörnig
    - jeder Test tut nur wenig
  - zahlreich
- Werkzeugunterstützung:
  - Java: JUnit
    - analog auch für viele andere Sprachen
  - Python: unittest, nose, pytest
- …und ferner für Mocking:
  - Java: jMock und andere
    - Achtung: Liste nicht aktuell
  - Python: unittest.mock und andere

(E1,E2) Technische Tests (Techies): Integrationstests

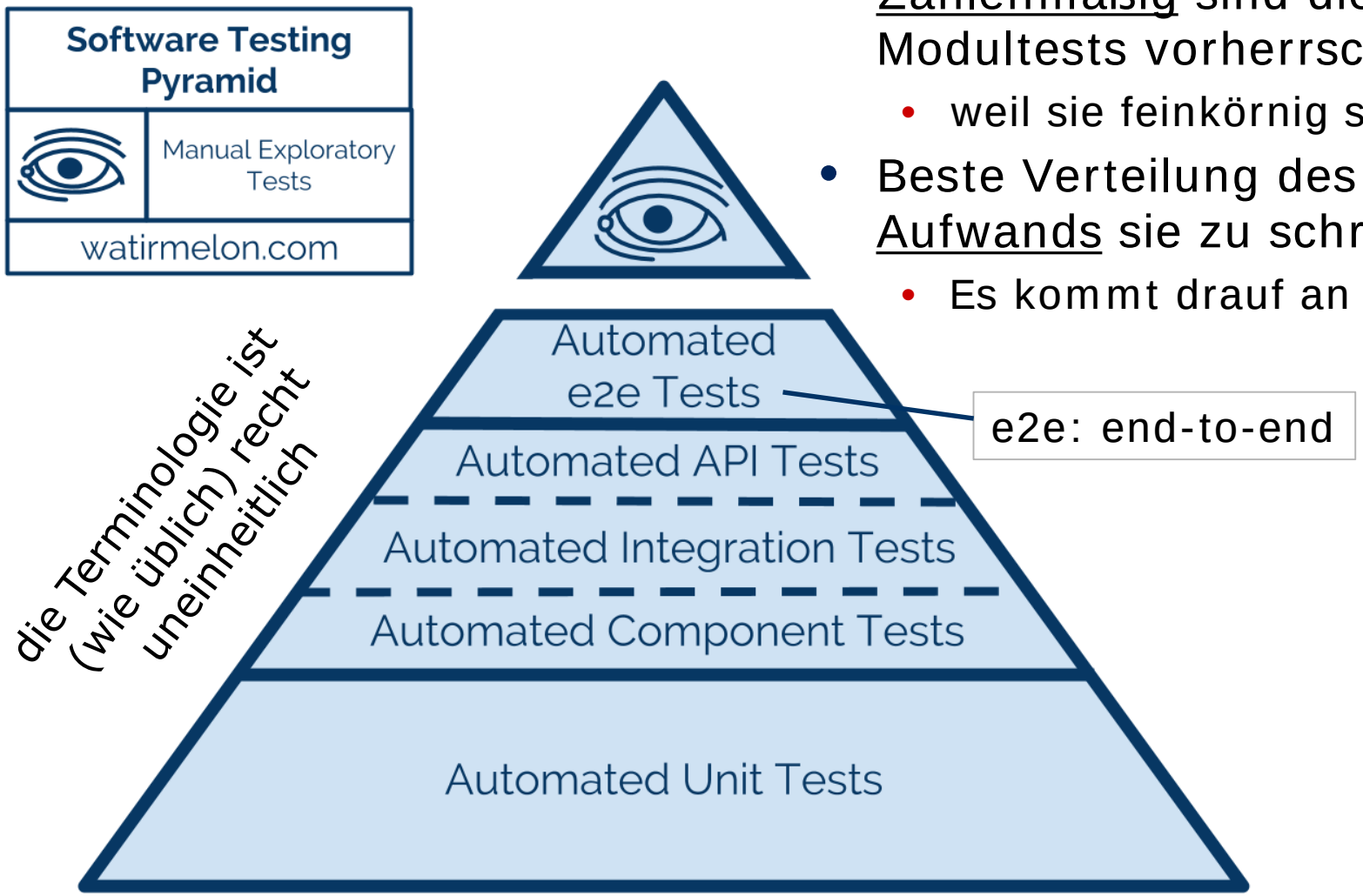
Freie Universität Berlin

- Ziel/Zweck:
  - Korrektes Zusammenspiel vieler Module prüfen
- Merkmale:
  - fast immer automatisiert
  - nur wenige Szenarios werden geprüft
    - im Vergleich zu den möglichen
    - relativ gesehen viel weniger als bei Modultests
  - Erfolgsfälle ("happy path") und einige Fehlerfälle
- Werkzeugunterstützung:
  - wie bei Modultest

(E1,E2) Systemtests, end-to-end-Tests

- Ziel/Zweck:
  - Das komplette System im Ganzen testen
    - insbesondere durch das GUI
    - mit allen Teilen und benutzten Fremdsystemen
- Merkmale:
  - oft ziemlich unpraktisch und aufwändig
  - deshalb evtl. nur elementare Testfälle ("smoke tests")
- Werkzeugunterstützung:
  - Selenium
    - wenn das GUI ein Web-GUI ist
    - andernfalls gibt es kommerzielle Werkzeuge
  - (Und wenn man gar kein GUI hat?
    - Na, dann halt anders)

(E1) Technische Tests (Techies): Anteile der Sorten? ("Testpyramide")



- Zahlenmäßig sind die Modultests vorherrschend
  - weil sie feinkörnig sind
- Beste Verteilung des Aufwands sie zu schreiben:
  - Es kommt drauf an

(E1,E2) Technische Tests (Techies): Leistungstests

- Ziel/Zweck:
  - Trends erkennen, wo ein System allmählich langsamer wird
    - oder schon zu langsam ist
- Merkmale:
  - Form ist meist Integrationstest
    - nicht Modultest
    - aber ggf. separat für bestimmte Funktionen
      - nicht vermischt wie beim Systemtest
- Werkzeugunterstützung:
  - wie bei Integrationstests oder
  - evtl. wie bei Systemtests oder
  - httperf
  - u.a.

(E1,E2) Geschäftsnutzentests (**Kunde**): Akzeptanztests

Freie Universität Berlin

- Ziel/Zweck:
  - Endbenutzer-relevante Funktionalität mit realistischen Abläufen im Ganzen prüfen
- Merkmale:
  - werden evtl. mit Englisch-artiger Syntax formuliert
    - und mit Adaptern in Programmabläufe umgesetzt
  - Doug Bradbury: "10 Ways to do Acceptance Testing Wrong"
- Werkzeugunterstützung:
  - Selenium
    - wenn es durchs GUI sein muss (pflegeaufwändig!)
  - FitNesse, Cucumber
    - pseudosprachlich-lesbare Integrationstests
  - u.a.

(E1,E2) Geschäftsnutzentests (Kunde): Leistungstests

- Ziel/Zweck:
  - Geschwindigkeits- und Kapazitätsanforderungen überprüfen
- Merkmale:
  - meist auf Ebene von Systemtests
  - verwenden meist große Datenmengen und/oder viele gleichzeitige Benutzer
- Werkzeugunterstützung:
  - Selenium
  - httperf
  - u.a.

Wozu dienen automatische Modul- und Integrationstests?

- **Risiko reduzieren!**
- Welche Risiken?:
  - Code ist defekt
  - Code wird später defekt
    - insbes. bei Refactoring
  - Entwurf ist ungünstig
    - Das Schreiben der Tests hilft, einen guten Entwurf zu finden
      - insbes. gut entkoppelt
  - Code ist schwer zu verstehen
    - Test dient als Dokumentation oder Spezifikation
- Nutzen also:
  - analytische QS
  - "Sicherheitsnetz"
    - und schnelles Debugging
  - Entwurfshilfe
    - Das klappt aber nur beim Test-First-Vorgehen richtig
  - "Specification by Example"

- Cool!
  - Also: je mehr Tests, desto besser?
  - Nein!:

(E4) Kosten und Risiken automatisierter Tests

1. Aufwand, sie zu schreiben

2. Aufwand, sie mit zu ändern
   - beim Refactoring
   - bei Anforderungsänderngn.
   - insbes. wenn eine Änderung viele Tests betrifft
     - das passiert umso mehr, wenn man redundante Tests hat

3. Zeitaufwand, sie ablaufen zu lassen
   - insbes. bei langsamen Tests
     - vor allem GUI-Tests!

4. Aufwand zum Verstehen
   - insbes. bei redundanten Tests

5. Verwirrung durch falsche Versagen
   - z.B. wg. Zeitbedingungen
   - z.B. weil Test veraltet

(E4) Eigenschaften guter Testsuites

- Akzeptanztests:
  - fördern Fokus auf benötigte Funktionalität !
  - sichern Geschäftsfunktionen gegen Regression ab !
    - so schlank und schnell wie möglich
  - sind Specification by Example !

! Bitte gründlich!

? Nach Bedarf viel oder wenig

- Modultests:
  - sind Dokumentation ?
  - prüfen Logik eines Moduls ?
    - lokale Logik oder
    - Aufrufe anderer Module
  - leiten den Entwurf zu hoher Entkopplung ?
    - SRP, TDA, DIP, IH

- Integrationstests:
  - prüfen Zusammenspiel von Modulen !
  - sichern interne Funktionen gegen Regression ab !
    - so schlank und schnell wie möglich
  - sind evtl. Dokumentation ?

Wie bekommt man selbsttestenden Code hin? (Reprise)

- Wie bekommt man
  - und zwar mit erträglichem Aufwand
  - selbsttestenden Code
  - mit wenig Testredundanz
  - und hoher Testgeschwindigkeit?
- Und zwar
  - sowohl für Verifikation
    - Testen gegen Spezifikation
  - als auch für Validierung
    - Testen gegen tatsächliche Anforderungen?

- Elemente (E1…E4):
  1. Testarten unterscheiden ✓
  2. Jeweils gut passende Werkzeugunterstützung bereit stellen ✓
  3. Methoden in Erwägung ziehen:
     1. Attrappen, Testisolation
     2. Teststile
     3. Testgetriebene Entwicklung
  4. Immer schön Maß halten! (✓)
     - Bei Anforderungen an Testabdeckung
     - Stilanforderungen
     - Isolation
     - Geschwindigkeitsanford.
     - TDD-Einsatz

Ende von Teil 1

E3.1: Methoden in Erwägung ziehen: Attrappen, Testisolation

Freie Universität Berlin

- Für manche Leute bedeutet "Modultest" sofort auch *"alle benutzten Module durch Attrappen ersetzen"*
  - oder: Feathers' Kriterien
  - Das ist aber nicht zwingend
  - Modultest beschreibt nur, wo das Erkenntnisinteresse liegt: Funktion dieses Moduls
- Aber Attrappen können **Vorteile** bieten:
  1. läuft schnell ab
  2. ist stets verfügbar
  3. Spy-ing anstatt nur Zustandsprüfung
- Diese müssen aber gegen die **Nachteile** abgewogen werden:
  - Attrappe kann falsch sein
    - → Illusionen
    - KISS??
  - Attrappe muss mit geändert werden
    - verletzt DRY
  - Injektion von Attrappen verkompliziert den Entwurf
    - verletzt evtl. KISS
    - besonders bei High-Ceremony-Sprachen

E3.1: Methoden in Erwägung ziehen: Attrappen, Testisolation

Freie Universität Berlin

Gründe für die Verwendung von Attrappen:

- **ZO**: Zielobjekt
- **A**: Attrappe des ZO
- [MockStub]: Unterscheide Stub&Fake von Spy&Mock!

- Als Entwurfshilfe
  1. outside-in-Entwicklung: das ZO existiert noch gar nicht (→ meist Mock/Spy)
  2. A erlaubt schnelleres Ausprobieren möglicher Zerlegungen und APIs

- Als Testhilfe
  3. Bereitstellung gewünschter Testinputs (→ Stub/Fake)
     - z.B. bei ext. Diensten, die zeitvariable Daten liefern
     - z.B. zum Testen von Fehlerbehandlung
  4. Schnellere Testausführung
  5. Spying (z.B. f. Cache)
- Purismus
  6. *"eine Unit muss im Test komplett isoliert werden"*
     - Unprofessioneller Grund!
     - Isolation hat einen Zweck
       - nur der taugt ggf. als Begründung
       - Wo nicht: nicht isolieren!

E3.1: Fallstudie Attrappen/Isolation: WP

- Realisiert einen Quant Fonds
  - Monatlich Wertpapiere aus einer festen Gruppe kaufen/verkaufen nach rein zahlenmäßigen Kriterien
  - Hier: nur historische Kurse, nur Kauf*entscheidung*, ETFs
- Aufbau:
  1. Lies WP-Liste aus Exceldatei
     - metadata.py
  2. Hole Kursdaten von Yahoo
     - quotesservice.py
  3. Werte Kriterien aus
     - portfolio.py
  4. Drucke Bericht m. Aktionen ggü. Vormonat (portfolio.py)
- Attrappen-Nutzung:
  - main.py: Spy ersetzt Metadata, Quotesservice, Portfolio
    - f. Grobentwurf, Schritt 1
  - metadata.py: --
    - sondern Testdaten-Dateien
  - quotesservice.py: --
    - Plausi-Assertions im Modul
    - Realtest: Vergleich mit Datenkonserve (wichtig!)
  - portfolio.py: Stubs f. Metadata und Quotesservice
    - Erzeuge maßgeschneiderte Testdaten programmatisch
    - Spys lohnen nicht, da Ablauf simpel

E3.2: Methoden in Erwägung ziehen: Teststile: Test-First vs. Test-Last

Freie Universität Berlin

- Vorteile von **Test-First**:
  - hilft beim Ausdenken passender und bequemer Schnittstellen
  - fördert Nachdenken über Funktionalität *vor* dem Verwickeln in Details
  - erlaubt frühes Feedback

- Vorteile von **Test-Last**:
  - spart Arbeit:
    - vermeidet Testrefactorings
    - vermeidet Attrappen, die man nur als Zwischenlösung braucht
  - ich darf einfach weiter implementieren wenn's gerade "flutscht"

Beides sind manchmal wichtige Gründe

E3.2: Methoden in Erwägung ziehen: outside-in vs. inside-out/middle-out

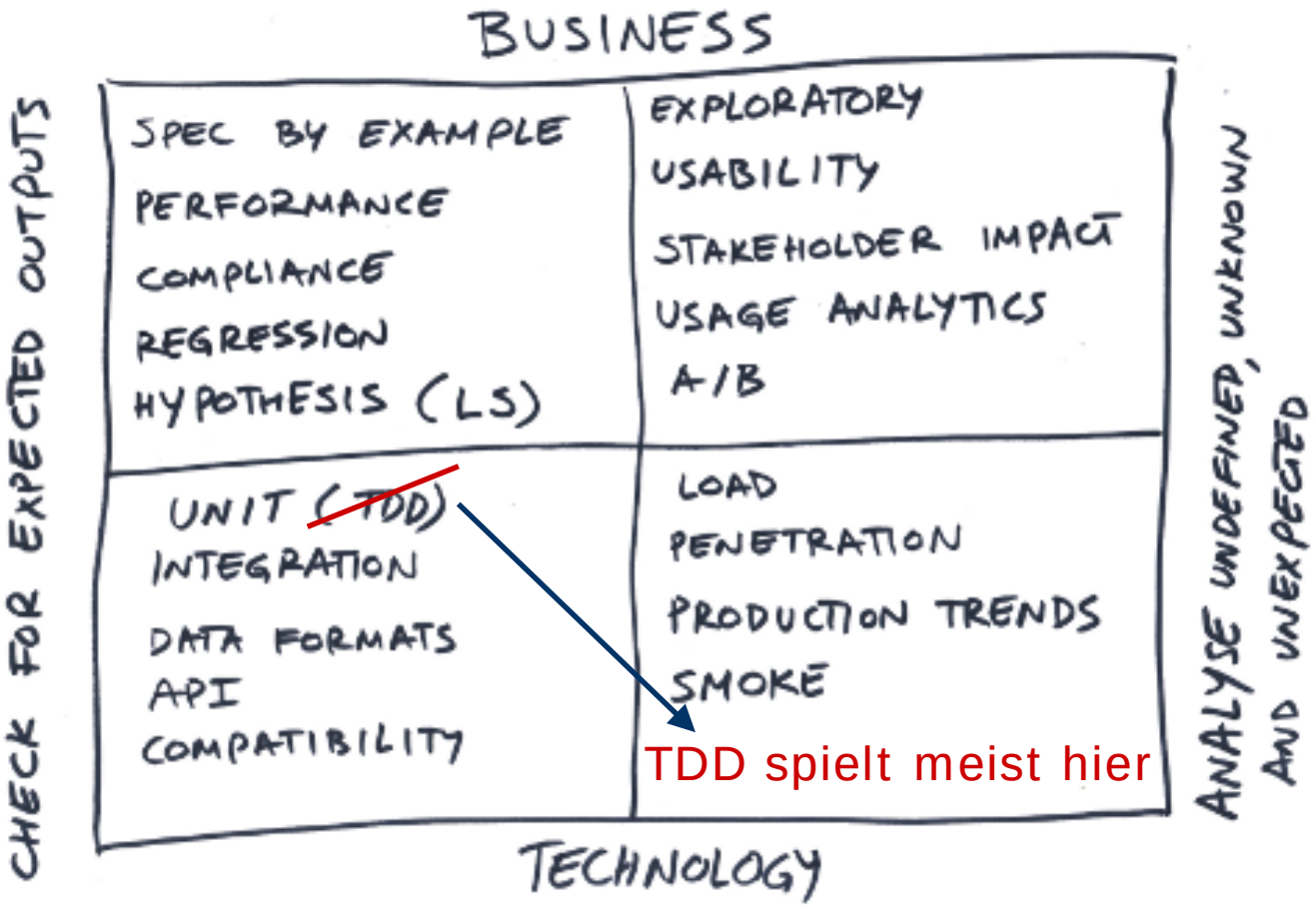
Freie Universität Berlin

- Outside-in (Top-Down):
  - Schreibe das zuerst, was von Außen aufgerufen wird: GUI oder Fassade
  - Realisiere jedes Kollaborationsobjekt als Mock oder Stub
    - "mockist style" (vs "classicist style")
  - Mache dann dort weiter
- Vorteil:
  - Jederzeit gute Orientierung beim Entwurf
  - Mockists: discovery testing
- Middle-out (Bottom-up):
  - Ergänze zuerst die nötige Funktionalität in den Domänenobjekten
  - Arbeite dich dann Richtung Aufrufer vor
- Vorteile:
  - Classicists: Viele Attrappen werden unnötig
  - Vermeidet, Geschäftslogik am falschen Platz zu haben
    - Mehrfachverwendung leichter zu erkennen

Test-First heißt nicht nur überprüfen, sondern Entwurf erkunden

Freie Universität Berlin

- Analyzing (Erkunden) vs. Checking (Überprüfen) siehe
  - http://www.satisfice.com/blog/archives/856



E3.2: Methoden in Erwägung ziehen: Teststile: classicist vs. mockist

Freie Universität Berlin

- Mockist-Entwicklungsstil:
  - Stets perfekte Isolation
    - Tests werden schnell
    - Die Attrappen sind klein und einfach
    - Defekte lassen nur wenige Tests versagen
      - nicht fremde Module auch noch mit
  - Stets Verhaltensprüfung
    - d.h. White-Box-Testen!
    - evtl. zusätzlich Ergebnisprüfung
      - Black-Box

- Classicist-Entwicklungsstil:
  - Attrappen nur wo nötig
    - Fast alle Tests prüfen auch Integrationsaspekte mit
    - Manche Attrappen können später wieder weggeworfen werden
    - Refactorings und Semantikänderungen betreffen weniger Einzelteile
      - weniger Attrappen
      - seltener die Tests in ihrer Struktur
  - Bevorzugt Ergebnisprüfung
    - Verhaltensprüfung nur zur Vereinfachung in schlimmen Fällen

Sehr gute Diskussion bei Fowler (wo sonst?): [MockStub]

E3.3: Methoden in Erwägung ziehen: Testgetriebene Entwicklung (TDD)

Freie Universität Berlin

- TDD steht für Test-Driven Development
  - vormals *Test-Driven Design*
    - das ist auch der bessere Name
- TDD ist die Idee, nicht nur Test-First zu arbeiten, sondern das in kleinen Schritten zu tun:
  - Code Testfall-für-Testfall entwickeln
  - Entwurf unterwegs entstehen lassen
  - Viel Refactoring machen, um eine sehr gute Endstruktur zu erhalten

TDD-Definitionen:

- Martin Fowler (hier):
  - *"Write a test for the next bit of functionality you want to add.*
  - *Write the functional code until the test passes.*
  - *Refactor both new and old code to make it well structured."*
- Kent Beck [XP2, Chap.7]:
  - *"Write a failing automated test before changing any code."*
    - (Mehr nicht, weil "Incremental Design" eine separate Praktik ist)

TDD-Definitionen (2):

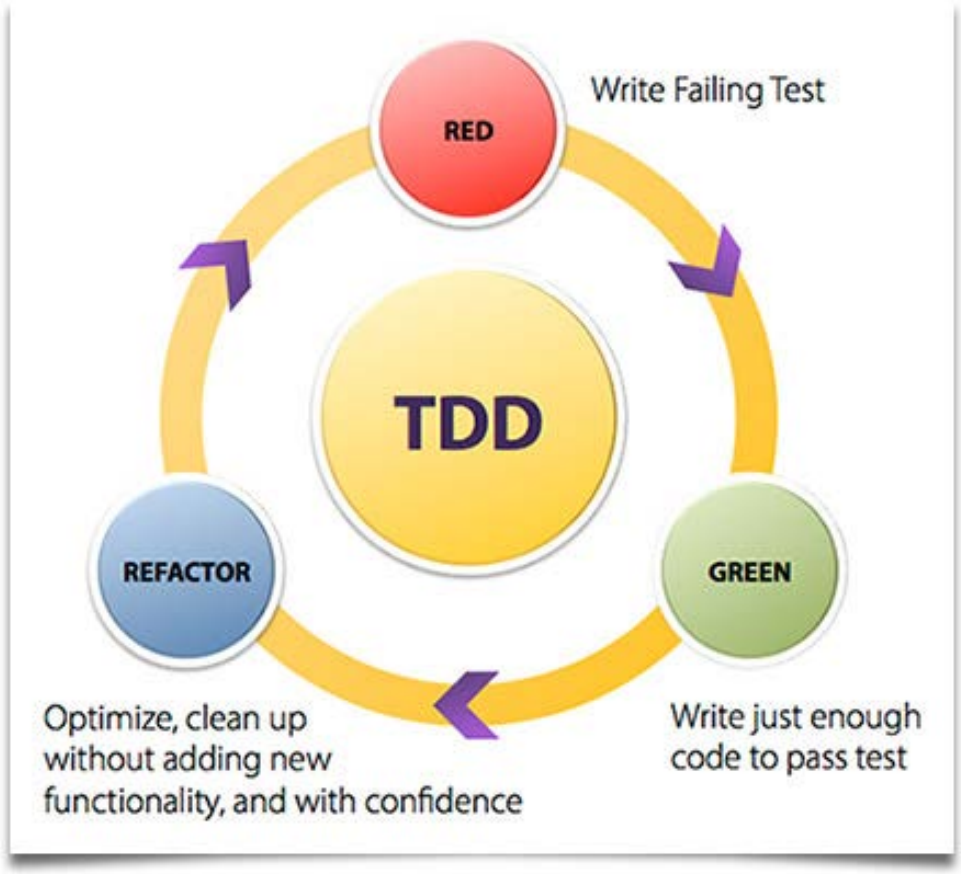
- Uncle Bob [ClC, Kap.9]:
  - *"You may not write production code until you have written a failing unit test.*
  - *You may not write more of a unit test than is sufficient to fail, and not compiling is failing.*
  - *You may not write more production code than is sufficient to pass the currently failing test.*
  - *These three laws lock you into a cycle that is perhaps thirty seconds long."*

- Anmerkungen:
  - Bessere Vorläuferquelle: [TDDlaws]. Dort hieß es noch "two minutes" und es waren Ausnahmen vorgesehen.
  - Nicht ganz seriös: Was ist mit Refactoring??

E3.3: Methoden in Erwägung ziehen: Testgetriebene Entwicklung (TDD)

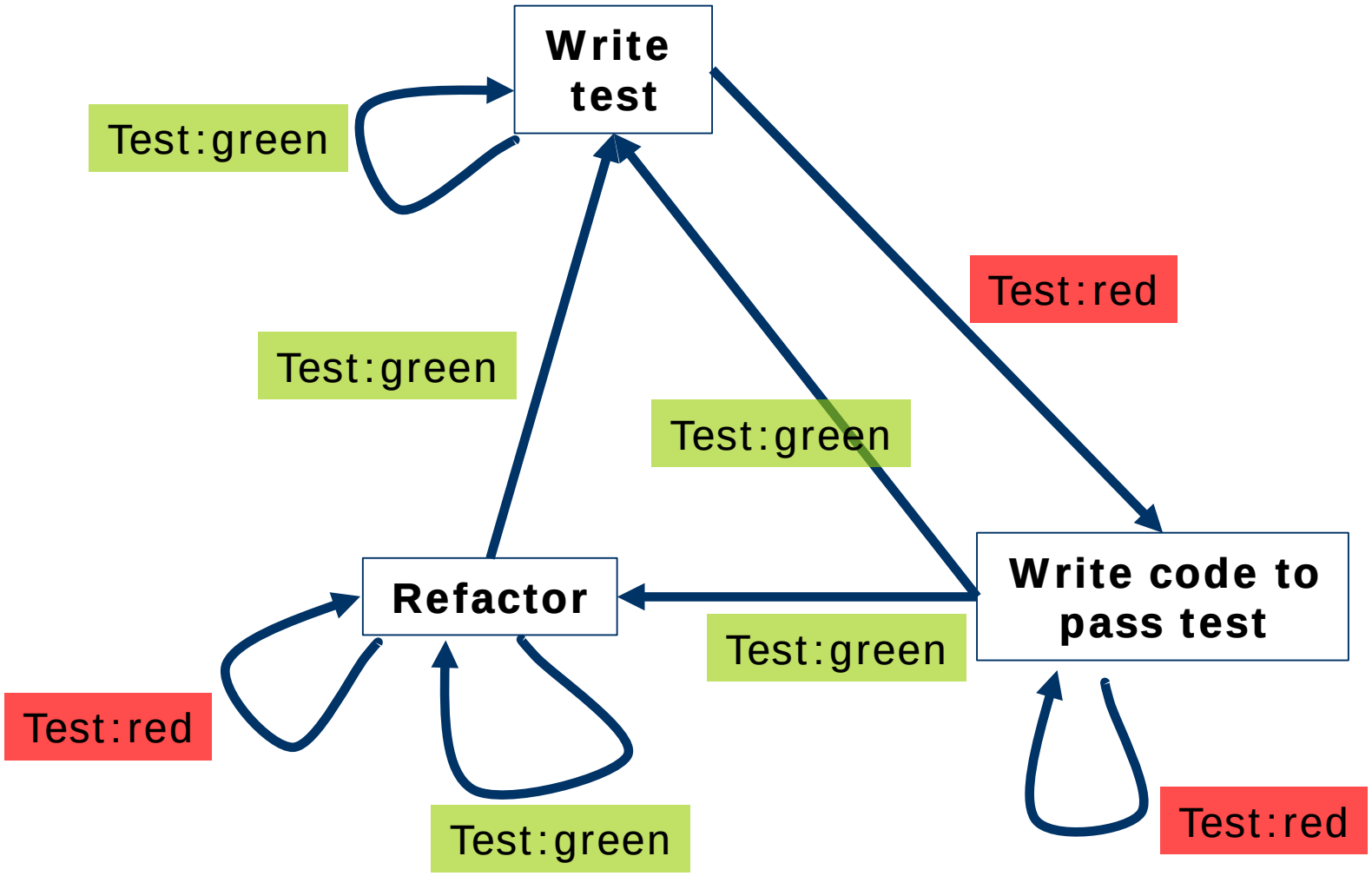
TDD-Definitionen (3):

- Die gängigste: "Red/Green/Refactor"



- Gut daran:
  - Betont die Test*ausführung*
  - Betont, dass die Entwurfsarbeit (Refactoring) so durch Tests abgesichert ist
- Aber: Irreführendes Bild!
- Schlecht daran:
  - Mischt Testergebnisse (Red, Green) mit Tätigkeiten (Refactor)
    - und zeigt die Ereigniskette nur unvollständig:

TDD: Red/Green/Refactor: So sieht's wirklich aus



Wie lernt man TDD?

- Nur durch Ausprobieren und Üben
- Anfangs in spielerischen "Trockenübungen" in einer ausdrücklichen Übungssituation
  - *"Dojo"*: Übungsraum
- Formen dafür:
  - "Randori": improvisiertes, freies Üben
  - "Kata": eigentlich: ritualisierter Ablauf
    - zum Perfektionieren der "Bewegungen"
    - Faktisch meist: Wort für die zu lösende Aufgabe

Ressourcen:

- Listen von Übungsaufgaben:
  - tddproblems
  - bei cyber-dojo.org
- Virtueller Übungsraum:
  - cyber-dojo.org
- Berliner Übungsraum:
  - Softwerkskammer Berlin
    ca. monatliche Treffen

Dojo
Randori
Kata

Wir lernen TDD!

- Per Gruppen-Randori mit Diskussion
  - Ich tippe
  - Sie sagen an
    - und wir diskutieren Alternativen
  - Red-Green-Refactor
  - Hauptziel ist Lernen
    - nicht schnell fertig werden

- Aufgabe:
  - decimal_to_roman.py
    - decimal_to_roman(1964) == "MCMLXIIII"
  - schwach subtraktiv, d.h. subtraktiv bei den Zehnern, aber nicht bei den Fünfern: IIII (nicht IV), aber IX (nicht VIIII) usw.
  - I=1, V=5, X=10, L=50, C=100, D=500, M=1000

E4: Immer schön Maß halten! TDD-Einsatz

Freie Universität Berlin

TDD passt nicht immer:

- Bob Martin 2014-01-27:
  - TDD muss erlernt werden
  - Man darf nicht für jede Methode einen Test erzwingen wollen
  - Die Architektur muss zuvor feststehen
- David Heinemeier Hansson (DHH) 2014-04-23 "TDD is dead. Long live testing":
  - TDD passt schlecht zur Entwicklung von Rails-Anwendungen
  - Model: Test direkt auf DB
  - sonst: Controllertests oder sogar Systemtests
  - Laufen langsamer, sind aber insgesamt sinnvoller
  - TDD für Entwurf scheint unnötig
- Riesenaufruhr und Diskussion
  - z.B. Ralf Westphal: Nutze es nur, wenn es passt
- Bob Martin 2014-04-30 "When TDD does not work":
  - separate code that needs checking (logic → TDD) from code that needs fiddling (UI → best tested manually)
  - (aber nicht nur "Logik" braucht "checking")

E4: Immer schön Maß halten! Stärken und Grenzen von TDD

Freie Universität Berlin

- Kent Beck 2014-06-02 "Learning About TDD":
  - "I'm puzzled by the limits of TDD
  - it works so well for algorithm-y, data-structure-y code. I love the feeling of confidence I get when I use TDD. I love the sense that I have a series of achievable steps in front of me
  - can't imagine the implementation? no problem, you can always write a test.
  - I recognize that TDD loses value
    - as tests take longer to run,
    - as the number of possible faults per test failure increases,
    - as tests become coupled to the implementation, and
    - as tests lose fidelity with the production environment.
  - How far out can TDD be pushed?
    Are there special cases where TDD works surprisingly well? Poorly?
    At what point is the cure worse than the disease?"

E4: Immer schön Maß halten! "TDD is dead"-Diskussion

- DHH, Kent Beck und Martin Fowler machten vom 9.5. bis 4.6.2014 fünf Hangouts dazu. Kernpunkte:
  - Verwechsle nicht TDD mit selbsttestendem Code
  - Die Passung von TDD hat mit Persönlichkeit und Denkstil zu tun
    - Beck hilft es gegen Ängste (1;7:20-7:52)
    - DHH schreibt lieber Lösungen hin (1;11:40-12:32)
  - Man braucht eine klare Spezifikation
    - Beck (1;13:22-15:34)
    - DHH (1;16:52-18:41)
- Evtl. muss man TDD gegen KISS abwägen
  - z.B. kann hohe Kopplung (→untestbar) im Controller die Verständlichkeit sehr erhöhen (DHH:2;23:09-24:49)
- TDD kann zu tollen Entwurfsideen zwingen…
  - Beck (2;26:50-27:51)
- …schafft das aber nicht immer
  - DHH (2;27:52-28:44)

E4: Immer schön Maß halten! Testabdeckung

- Kent Beck "I get paid for code that works, not tests."
- Beachte:
  - 100% Anweisungsabdeckg. heißt nicht sofort "gut getestet"
    - nur Assertions sind Tests!
  - Die letzten paar Prozent sind evtl. einfach zu schwierig zu bekommen
    - also zu teuer: weglassen
  - Manche Sachen macht man selten falsch
    - oder nur anfangs: Review reicht!
  - Grobe Fehler findet auch ein Integrationstest
- Hauptnutzen von Tests ist das Absichern von Refactorings
  - Fowler (4;23:32-24:38)
- Ohnehin ersetzen autom. Tests manuelle QA nicht komplett
  - DHH (2;21:52-23:59)

(Wer es polemisch mag:

- DHH über over-testing
  - TSA: Transportation Security Administration
    - siehe deren Effektivität)

E4: Immer schön Maß halten! Isolation

- Beck: (1;20:48-21:38/22:20):
  - My designs must be flexible, nice, and testable
  - I usually find *some* way to get feedback, with the real stuff or without.
  - Too much mocking may expose so much implementation as to make refactoring near-impossible.
- Fowler (1;23:14-24:13):
  - Don't confuse unit testing with isolation
  - There are styles that work well without much isolation
  - "I hardly ever use mocks"
    - "but I know good people who do"

All diese Videoschnipsel bitte mal in Ruhe ansehen und durchdenken.

E4: Immer schön Maß halten! Geschwindigkeitsanforderungen

Freie Universität Berlin

- DHH: "Slow DB test fallacy"
  - (Anfangs polemisch, dann aber vernünftig)
  - Unit tests sind auch durch die DB meist schnell genug
    - Warum also den Entwurf verkomplizieren?
  - Nach lokalen Änderungen muss nicht die komplette Suite ablaufen.

Standardfall

- Gary Bernhardt: "TDD, Straw Men, and Rhetoric"
  - Antwort auf DHH
  - (Extrem beim Zeitwunsch, mindestens einmal unehrlich, balanciert ansonsten)
  - "I want my feedback to be so fast that I can't think before it shows up": 300msec
  - "It means that the flow of my thoughts never breaks"

spezieller Fall

Hausaufgabe

1. TDD Einüben:

- Zwei weitere Übungsprogramme von cyber-dojo.org (Auswahl nach eigenem Geschmack) im TDD-Stil entwickeln
  - in Paararbeit
    - jetzt Partner suchen
  - mit Ping Pong Pair Programming
- Dabei notieren:
  - Probleme,
  - Aha-Erlebnisse,
  - Erfolgserlebnisse.

2. Diesen Foliensatz ab E3.2 nochmal durcharbeiten.

Danke!

https://eev.ee/blog/2016/08/22/testing-for-people-who-hate-testing/

Domain-driven Design: DDD resources

Freie Universität Berlin

- DDD: Entity Injection and Mocking Time
  - http://blog.jonathanoliver.com/ddd-entity-injection-and-mocking-time/
- http://programmers.stackexchange.com/questions/205731/should-we-mock-entities-and-value-objects-when-doing-ddd
- http://stackoverflow.com/questions/2833422/how-to-keep-your-unit-tests-simple-and-isolated-and-still-guarantee-ddd-invarian
- http://www.taimila.com/?p=1516 (DDD and testing strategy)