

Programmierkonzepte in der Physikalischen Chemie

19. Testautomatisierung und testgetriebene Entwicklung

Albert-Ludwigs-Universität Freiburg



**UNI
FREIBURG**

Dr. Till Biskup

Institut für Physikalische Chemie
Albert-Ludwigs-Universität Freiburg
Wintersemester 2018/19



Zentrale Aspekte

- ❏ Automatisierung ist Voraussetzung für häufiges Testen – und für Refactoring und Verbesserung der Codequalität.
- ❏ Testautomatisierung hat tiefgreifenden Einfluss auf den Entwurf des zu testenden Codes (u.a. Modularität).
- ❏ Testgetriebene Entwicklung stellt die Reihenfolge auf den Kopf und sorgt für hohe Testabdeckung.
- ❏ Hohe Testabdeckung durch automatisierte Tests erhöht das Vertrauen in Software.
- ❏ Testgetriebene Entwicklung führt zu Tests, die den Code und seine Spezifikationen dokumentieren.

Grundüberlegungen zur Testautomatisierung

Grundregeln testgetriebener Entwicklung

Auswirkungen testgetriebener Entwicklung

Wichtige Aspekte von Tests

“ *Most testing should be done automatically.
It's important to note that by “automatically” we mean that
the test results are interpreted automatically as well.*

– Hunt und Thomas

- ▶ Automatisierung setzt ausführbare Tests voraus.
 - Ergebnis durch Vergleich mit der Spezifikation
 - Code muss Überprüfbarkeit mit Spezifikation ermöglichen.
 - Testfälle formalisieren und kanonisieren Spezifikationen.
- ▶ Ziel: übersichtliche Statistik über Testergebnisse
 - Im Erfolgsfall genügt die Zahl durchlaufener Tests.
 - Nur im Fehlerfall sind mehr Informationen notwendig.

Grundstruktur eines automatischen Tests

Gilt für Unittests und Integrationstests gleichermaßen.

Listing 1: Grundstruktur automatisierter Tests (Pseudocode)

```
1 function test_functionName
    exp = get_expected()
3   obs = functionName(arguments)
    assert exp == obs
```

Drei Teile

- ▶ Erwartung
 - bekanntes, gewünschtes Ergebnis des Funktionsaufrufs
- ▶ Beobachtung
 - tatsächliches Ergebnis des Funktionsaufrufs
- ▶ Vergleich von Erwartung und Beobachtung
 - Boolescher Wert

- ▶ Es gibt zwei Konzepte von Gleichheit.
 - Identität: identisches Objekt (im Speicher)
 - Äquivalenz: gleicher Wert/Inhalt
- ▶ Spezifikationen zielen meist auf Äquivalenz
 - Meist werden Werte/Ergebnisse miteinander verglichen.
- ▶ Numerische Werte sind häufig Näherungen.
 - Alles außer Ganzzahlen lässt sich nicht exakt abbilden.
 - Äquivalenz führt meist nicht zum Ziel.
 - Überprüft werden sollte immer ein Intervall.
 - Die Repräsentation von Zahlen ist architekturabhängig.
- ▶ Die Genauigkeit ist meist abhängig vom Zahlenwert.
 - Genauigkeit über Funktion berechnen lassen

► Zielstellung

- Tests sollten vollständig unabhängig voneinander sein.
- Tests sollten ihre Umgebung immer genauso verlassen, wie sie sie vorgefunden haben.

► Realität

- Tests brauchen oft eine ganz bestimmte Umgebung.
- Tests verändern ihre Umgebung mitunter.

► Lösung

- Vorbereiten und Aufräumen in Funktionen auslagern
- minimiert Code-Doppelungen

☞ Auslagern hat Vor- und Nachteile bzgl. Lesbarkeit.

☞ Aufräumen sollte *immer* stattfinden.

- ▶ Zielstellung
 - Tests sollten von jedem jederzeit einfach aufrufbar sein.
 - Tests sollten einfach zu schreiben sein.
 - Statistik als Zusammenfassung eines Testdurchlaufs
- ▶ Lösung: automatisierte Test-Frameworks
 - sorgen für Konsistenz
 - erleichtern das Schreiben und Ausführen von Tests
 - stellen Setup- und Teardown-Funktionalität bereit
- ▶ bekannteste Lösung für Unittests: xUnit-Framework
 - für viele Sprachen verfügbar
- ☞ Automatisierte Tests sollten mit Hilfe eines Test-Frameworks geschrieben werden.

Listing 2: Beispiel für Setup- und Teardown-Funktionalität

```
class TestReporter(unittest.TestCase):
2     def setUp(self):
        self.report = report.Reporter()
4         self.template = 'test_template.tex'

6     def tearDown(self):
        if os.path.exists(self.template):
8             os.remove(self.template)

10    def test_render_with_template(self):
        with open(self.template, 'w+') as f:
12            f.write('')
        self.report.template = self.template
14    self.report.render()
```

- ➡ Setup: Initialisierung eines Objektes
- ➡ Teardown: Aufräumen auf dem Dateisystem

“ *Once a human tester finds a bug,
it should be the last time a human tester finds that bug.
The automated tests should be modified to check
for that particular bug from then on [...]*

*[...] we just don't have the time to go chasing after bugs
that the automated tests could have found for us.
We have to spend our time writing new code—
and new bugs.*

– Hunt und Thomas

- ☛ (Verifizierte) Fehler grundsätzlich in Tests verwandeln.
 - insbesondere im Zusammenhang mit einer Bugverwaltung

Zwei Grundregeln

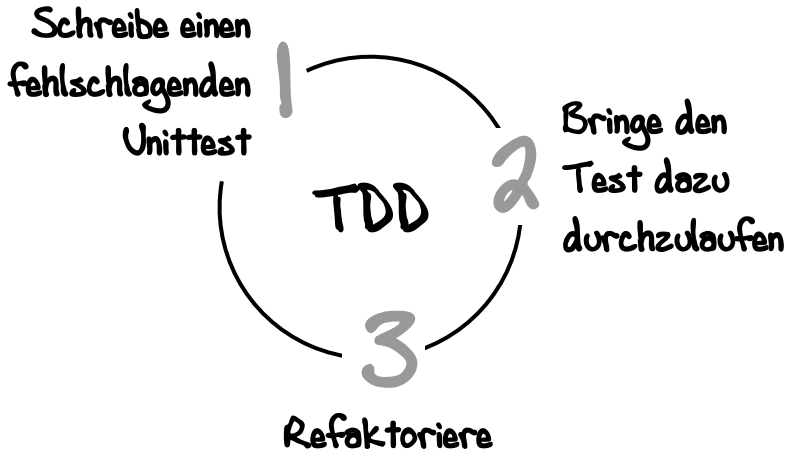
- ▶ Schreibe neuen Code nur und erst dann, wenn ein automatisierter Test fehlschlug.
- ▶ Eliminiere Code-Doppelungen.

Der Ablauf testgetriebener Entwicklung

- 1 Rot – fehlschlagender Test
 - 2 Grün – Test läuft erfolgreich durch
 - 3 Refactoring – entstandene Doppelungen eliminieren
- 👉 Zielstellung: sehr kurze Zyklen von wenigen Minuten Dauer

Testgetriebene Entwicklung (TDD)

Der Zyklus testgetriebener Entwicklung



Die drei Regeln von TDD nach Robert Martin

- ▶ kein Produktivcode ohne fehlschlagenden Test
 - Entwicklung beginnt mit dem Schreiben eines Tests.
 - Tests so schreiben, als ob es die Funktionalität schon gäbe.
- ▶ minimaler Testcode
 - nicht mehr als nötig, damit der Test fehlschlägt
 - Nicht Kompilieren zählt als Fehlschlag.
- ▶ minimaler Produktivcode
 - nicht mehr als nötig, um den Test erfolgreich zu durchlaufen
 - Vortäuschen der richtigen Antwort ist (anfangs) erlaubt.
- ☞ immer nur implementieren, was *wirklich* gebraucht wird, um alle Tests erfolgreich zu durchlaufen („YAGNI“)

Strategien, Tests zum Erfolg zu führen

Wie kommt man schnell dazu, Tests erfolgreich durchzulaufen?



Ziel des TDD-Entwicklungszyklus'

- ▶ kurze Zyklen: Test schreiben, Fehlschlag, Erfolg

Drei Strategien zum Erfolg

- ▶ Vortäuschen (*fake it*)
 - (temporär) direkte Rückgabe des Erwartungswertes
- ▶ offensichtliche Implementierung
 - hinreichend kleine Probleme und ausreichende Erfahrung
- ▶ Triangulierung
 - mehr als ein Test, unterschiedliche Erwartungswerte
- ☞ Erfolg des Tests ist Voraussetzung für Refactoring

Warum testgetrieben entwickeln?

Kleine Schritte machen Mut, das große Problem lösen zu können.



- ▶ Reale Probleme sind schwierig.
 - Häufig ist die Lösung nicht offensichtlich.
 - Oft wird zusätzlicher, unnötiger Code geschrieben – weil man ihn brauchen *könnte*.
- ▶ Testgetriebene Entwicklung geht in kleinen Schritten vor.
 - Ein Test spezifiziert ein Verhalten einer Funktion.
 - Erst wenn der Test erfolgreich durchläuft, wird der nächste Test geschrieben (und Code verbessert).
- ▶ Erfolgreich durchlaufende Tests geben Sicherheit.
 - Tests und eine hohe Testabdeckung sind Voraussetzung für die Verbesserung von Code.
 - Durchlaufende Tests sind ein Erfolgserlebnis.

Warum testgetrieben entwickeln?

Kleine Schritte geben Sicherheit.



“ *Once we get one test working, we know it is working, now and forever. We are one step closer to having everything working than we were when the test was broken.*

*Now we get the next one working,
and the next, and the next.*

*By analogy, the tougher the programming problem,
the less ground that each test should cover.*

– Kent Beck

- ☛ Die Größe der Schritte hängt vom konkreten Problem und der Erfahrung ab – die Tendenz ist eindeutig: *klein*.
- ☛ Wenn ein Problem schwerer als gedacht ist, sollten die Schritte entsprechend klein werden.

“ *Code without tests is bad code. It doesn't matter how well written it is; it doesn't matter how pretty or object-oriented or well-encapsulated it is. With tests, we can change the behavior of our code quickly and verifiably. Without them, we really don't know if our code is getting better or worse.*

– Michael C. Feathers

- ☛ Tests sind unverzichtbar für zuverlässige Software.
 - Konsens zumindest in der Softwareentwicklung
- ☛ Testgetriebene Entwicklung erzwingt Tests.
 - Tests werden *vor* dem Produktivcode geschrieben.

► Fokus

- YAGNI: Häufig wird unnötiger Code geschrieben.
- Tests sind explizite (und kompilierende) Spezifikationen.
- Jede Funktionalität wird durch einen Test eingefordert.

► schwache Kopplung und starke Kohäsion

- Schwer zu testender Code weist auf ein Designproblem hin.
- starke Kohäsion: Jede Funktion hat genau eine Aufgabe.
- Schwach gekoppelter, kohäsiver Code ist leicht zu testen.

► Vertrauen

- Nicht funktionierender Code ist wenig vertrauenswürdig.
- Sauberer, funktionierender, automatisch getesteter Code sorgt für Vertrauen – auch vonseiten anderer Entwickler.

► Flexibilität

- Tests sind ein Sicherheitsnetz für Veränderungen.
- Verbesserungen am Code sind gefahrlos möglich.

► Dokumentation

- Tests liefern Beispiele für den Aufruf von Funktionen.
- Tests kompilieren und sind daher immer aktuell.

► minimales Debugging

- Kurze Zyklen schränken den Bereich für Fehler ein.
- Wichtig: Regelmäßig/häufig *alle* Tests laufen lassen.

► besseres Design

- TDD fördert gutes Design: entkoppelt, modular, flexibel.
- Aber: TDD ist keine Garantie für gutes Design.

TDD testet implizit die Tests

Eine unerwartete Lösung zu einem unlösbaren Problem



“ *Technically, one of the biggest benefits of TDD nobody tells you about is that by seeing a test fail, and then seeing it pass without changing the test, you're basically testing the test itself. If you expect it to fail and it passes, you might have a bug in your test or you're testing the wrong thing. If the test failed, and now you expect it to pass, and it still fails, your test could have a bug, or it's expecting the wrong thing to happen.*

– Roy Osherove

☞ TDD umgeht die unendliche Regression, Tests zu testen.

Was macht einen guten Test aus?

Tests sind mindestens so wichtig wie der Produktivcode.

“ *If you don't keep your tests clean, you will lose them.
And without them, you lose the very thing
that keeps your production code flexible.*

– Robert C. Martin

▶ Lesbarkeit

- vielleicht noch wichtiger als in Produktivcode
- Kriterien: Klarheit, Einfachheit, Ausdrucksstärke

▶ Übersichtlichkeit

- Jeder Test sollte *genau ein* Konzept abtesten.

☞ Tests sind kein „Code zweiter Klasse“. Die Kriterien für „sauberen Code“ gelten hier mindestens genauso.

Tests sollten...

- F** schnell durchlaufen (*Fast*).
 - Tests sollten häufig aufgerufen werden.
- I** unabhängig voneinander sein (*Independent*).
 - Nur die Unabhängigkeit gewährleistet einfache Diagnose.
- R** in jeder Umgebung wiederholbar sein (*Repeatable*).
 - Tests sind Voraussetzung für gefahrlose Änderungen.
- S** einen Booleschen Rückgabewert haben (*Self-Validating*).
 - Voraussetzung für Objektivität und schnellen Durchlauf
- T** zeitnah zum zu testenden Code entstehen (*Timely*).
 - Tests zuerst schreiben führt zu testbarem Produktivcode.

TDD ist kein Allheilmittel

Es kommt auf den richtigen und korrekten Einsatz an



“ *It's a double-edged sword, and many people find this out the hard way.*

– Roy Osherove

- ▶ mögliche Folgen des korrekten Einsatzes
 - höhere Codequalität
 - geringere Fehlerzahl
 - erhöhtes Vertrauen in den Code
 - kürzere benötigte Zeit für das Finden von Fehlern
 - Verbesserung des Entwurfs des Codes
- ▶ mögliche Folgen des falschen Einsatzes
 - Frustration
 - Verringerung der Codequalität

- ▶ Wissen, wie man gute Tests schreibt
 - Tests sollten wartbar, lesbar und vertrauenswürdig sein.
 - Tests zuerst zu schreiben, reicht nicht aus.
- ▶ Tests zuerst schreiben (vor dem Produktivcode)
 - Tests zuerst zu schreiben hat entscheidende Vorteile.
 - Les- und wartbare Tests im Nachhinein bringen weniger.
- ▶ Tests gut entwerfen
 - Gutes Design ist eine Aufgabe für sich.
 - TDD führt nicht *per se* zu gutem/besserem Design.
- ☞ Tipp: nicht alle Kompetenzen gleichzeitig erwerben wollen
 - führt meist zu Frustration und ist wenig zielführend



Zentrale Aspekte

- ❏ Automatisierung ist Voraussetzung für häufiges Testen – und für Refactoring und Verbesserung der Codequalität.
- ❏ Testautomatisierung hat tiefgreifenden Einfluss auf den Entwurf des zu testenden Codes (u.a. Modularität).
- ❏ Testgetriebene Entwicklung stellt die Reihenfolge auf den Kopf und sorgt für hohe Testabdeckung.
- ❏ Hohe Testabdeckung durch automatisierte Tests erhöht das Vertrauen in Software.
- ❏ Testgetriebene Entwicklung führt zu Tests, die den Code und seine Spezifikationen dokumentieren.