

# Automatisierte Softwaretests

Thomas Lotze

25. Februar 2010

# Gliederung

- 1 Motivation
- 2 Minimalbeispiel
- 3 Techniken
- 4 Helfer
- 5 Ausblick

# Software enthält Fehler

*„Wenn Debugging der Vorgang ist, Fehler aus Software zu entfernen, dann ist Programmieren der Vorgang, Fehler einzubauen.“*

- Flüchtigkeitsfehler (z.B. Syntax)
- vorhersehbare (z.B. Verarbeitung von Nutzereingaben)
- unoffensichtliche (z.B. algorithmische)

# Komplexe Software hat komplizierte Fehler

- mehr potentielle Fehlerstellen
- mehr Code, in dem der Fehler zu suchen ist
- Zusammenspiel mehrerer Stellen im Code
- komplexere Strukturen (Entkopplung)
- Fehler von komplexerer Natur

# Weiterentwicklung macht es nicht besser

- erzeugt neue Fehler
- bringt behobene Fehler zurück
- macht den Code komplexer
- Wissen über Zusammenhänge geht verloren
- Umstrukturierung (Refactoring): reines Einbauen von Fehlern

# Wissen festhalten

## Absicht des Programmierers dokumentieren

- Fähigkeiten des Gesamtsystems (Features)
- Benutzung von Schnittstellen
- Verhalten von Einzelteilen (Implementierungsdetails)

## Wissen über Fehler bewahren

- vorhersehbare Fehler berücksichtigen
- aus aufgetretenen Fehlern lernen (Regressionen)

# Wissen anwenden

## Software testen

- beabsichtigtes Verhalten an Beispielen sicherstellen
- frühere Fehlersituationen nachvollziehen, Fehler ausschließen

## Tests automatisieren

- erwartetes Verhalten genau beschreiben
- Abläufe beliebig oft genau wiederholen
- Tests als Entwicklungswerkzeug: schnell und einfach ausführen

# Grenzen von Softwaretests

- Aussage nur über einzelne Fehler, nicht über Fehlerfreiheit
  - Tests sind abhängig von repräsentativen Beispielen
  - Autor ist blind gegenüber eigenen Programmierfehlern
- Testabdeckung aller Codepfade nicht möglich
- Korrektheit des Codes nicht bewiesen

# Unser Testkandidat

Sortierfunktion für Objekte mit verschiedenen Sortierschlüsseln

- Schnittstelle: `multisort(values, keys) → values`
- Eingabewerte, Schlüssel: iterierbar, Ausgabewerte: Liste
- Schlüssel: Funktion, Objekt → sortierbares Merkmal
- Implementierung:
  - Sortieralgorithmus nach einem Schlüssel
  - Sortierung auf mehrere Schlüssel anwenden

# Code

```
def multisort(values , keys):  
    for key in reversed(keys):  
        by_key = dict((key(v), v) for v in values)  
        values = []  
        while by_key:  
            values.append(by_key.pop(min(by_key)))  
return values
```

# Beispielaufruf

```
>>> obj_a = {1: 'foo ', 2: 'bar '}
```

```
>>> obj_b = {1: 'baz ', 2: 'quux '}
```

```
>>> def key_1(obj): return obj[1]
```

```
>>> def key_2(obj): return obj[2]
```

```
>>> multisort([obj_a, obj_b], [key_1, key_2])  
[{'1: 'baz ', 2: 'quux '}, {'1: 'foo ', 2: 'bar '}]
```

# Testcode

```
import multisort , unittest

class MultisortTestCase(unittest.TestCase):
    def test_simple(self):
        obj_a = {1: 'foo' , 2: 'bar' }
        obj_b = {1: 'baz' , 2: 'quux' }
        def key_1(obj): return obj[1]
        def key_2(obj): return obj[2]
        result = multisort.multisort(
            [obj_a , obj_b] , [key_1 , key_2])
        self.assertEqual([obj_b , obj_a] , result)

unittest.main()
```

# Software wehrt sich gegen Tests

- Unvorhersagbarkeit (z.B. Zufallswerte, äußere Einflüsse)
- Einheiten nicht klar abgegrenzt, interne Datenstrukturen
- Einheiten nicht einzeln ausführbar (enge Kopplung, Black box)
- Zusammenspiel mit fremden Systemen (Einrichtung, Isolation)
- schwer nachzustellende Situationen (z.B. Netzfehler)

# Gegenmaßnahmen

- Software so schreiben, daß sie sich testen läßt
  - Code-Struktur
  - Abhängigkeitsinjektion
- beim Test prinzipielle Hindernisse mit Technik erschlagen
  - Attrappen (Mock-Objekte)
- dabei Wissen über Tests von Produktionscode fernhalten

# Zwei Fliegen mit einer Klappe

Guter Programmierstil führt zu leichter testbarem Code:

- Einheiten abgrenzen, um sie einzeln ansprechen zu können
- Funktionen kapseln, einfache Daten austauschen
- globalen Zustand vermeiden

# Beispiel

```
def multisort(values , keys):  
    for key in reversed(keys):  
        values = singlesort(values , key)  
    return values  
  
def singlesort(values , key):  
    by_key = dict((key(v), v) for v in values)  
    values = []  
    while by_key:  
        values.append(by_key.pop(min(by_key)))  
    return values
```

# Hintertüren einbauen

Stilmittel der losen Kopplung auf die Spitze treiben:

- Wissen über Umgebung aus Einheiten fernhalten
- Kommunikationspartner von außen übergeben
- Testattrappen direkt übergeben, statt Umgebung zu ersetzen

# Beispiel

```
def multisort(values, keys, singlesort_impl):
    for key in reversed(keys):
        values = singlesort_impl(values, key)
    return values
```

```
def test_isolated(self):
    <SETUP>
    def singlesort_impl(values, key):
        return sorted(values, key=key)

    result = multisort.multisort(
        [obj_a, obj_b], [key_1, key_2],
        singlesort_impl)
    self.assertEqual([obj_b, obj_a], result)
```

# Black boxes aufbohren

Die Umgebung der getesteten Einheit mit Testcode unterwandern:

- Verhalten beobachten (z.B. Funktionsaufrufe, Netzverkehr)
- Code gezielt mit Daten füttern
  - Zufall ausschließen, äußere Einflüsse kontrollieren
  - Systeme simulieren, die im Test nicht verfügbar sind
  - Ausnahmesituationen simulieren: Fehler, seltene Ereignisse
- zu testenden Code einsperren

Beispiel-Implementierung: Minimock

# Mehr Automatisierung als mit unittest

## Ziele:

- Tests finden, Doc-Tests in den Quelldateien erkennen
- Ausführungsumgebung vorbereiten, Tests isolieren
- Ausführung kontrollieren (Reihenfolge, Parallelisierung)
- Ergebnisse aufbereiten
- Test-API bereitstellen (z.B. Browser ansteuern)

## Beispiele:

- `setup.py test`, `zope.testing`, `nose`
- `manuel`
- `Selenium`

# Der Testrunner von zope.testing

```
$ easy_install zope.testing
```

```
$ zope-testrunner --path . --tests-pattern '^test'
```

```
$ zope-testrunner --path . --tests-pattern '^test' -
```

# Getestete Dokumentation, dokumentierte Tests

## Tests in Doc-Strings von Python-Code

- Beispiel: `tl.geodrawing.controller`

## Tests in eigenen Textdateien

- einfaches Beispiel: `tl.geotrack, util.txt`
- Beispiel mit Grafik: `tl.geodrawing, controller.txt`

# Webanwendungen im Browser testen

- Selenium RC
  - Beispiel: gocepts Webmailer

# Was sonst noch schiefgehen kann

- Fuzzing (Tests mit zufällig erzeugten Eingangsdaten)
- Belastungstests (gegen DoS, Race-Conditions, Speicherlecks)
- Leistungstests (z.B. Rechenzeit, Antwortzeit, Datendurchsatz)
- Profilierung (z.B. Rechenlast, Speicherbedarf, Netzlast)
- Codequalität (z.B. Stil, Importe)

# Stetige Integration

- Arbeit aller Entwickler unter gemeinsamer Revisionsverwaltung
- jederzeit Stand bekannter Qualität bzw. ohne bekannte Fehler
- automatische Tests bei jedem Integrationsschritt
- Zustand des Codes für alle leicht einsehbar (rot/grün)
- Werkzeuge: z.B. buildbot, Hudson

# Testgetriebene Entwicklung

- testen, entwickeln, umstrukturieren
- neuer versagender Test vor jeder Änderung
- Extreme programming