Wissenschaftliche Softwareentwicklung

16. Tests

Till Biskup
Physikalische Chemie
Universität Rostock
08.12.2023

Zentrale Aspekte

- Überprüfung ist ein Kernaspekt sowohl von Wissenschaft als auch von moderner Softwareentwicklung.
- Überprüfung wissenschaftlicher Software ist entscheidend, weil sie den Erkenntnisgewinn selbst betrifft.
- Der größte Teil wissenschaftlicher Software ist unzureichend oder gar nicht getestet.
- Tests können nur die Existenz von Fehlern beweisen, niemals deren Abwesenheit.
- Tests sollten so früh wie möglich, so oft wie möglich und so automatisiert wie möglich durchgeführt werden.

Überprüfbarkeit als zentrales Kriterium

Gemeinsamkeit von Wissenschaft und Softwareentwicklung

> *Testing is so central to both the scientific method and modern software development that many computational scientists consider it a moral failing for a scientific program not to include tests.*
>
> – Scopatz und Huff

Zwei Einheiten – aufgrund der Bedeutung der Thematik

1. Einführung in die Thematik
   - Bedeutung für die Softwareentwicklung
   - Terminologie, praktisches Vorgehen und allgemeine Regeln
2. Testautomatisierung und testgetriebene Entwicklung
   - konkrete Beispiele für Unittests
   - Bedeutung der testgetriebenen Entwicklung für Codequalität

Scopatz und Huff: Effective Computation in Physics, O'Reilly, Sebastopol 2015, S. 404

Übersicht

Motivation: zwingende Voraussetzung für Wissenschaftlichkeit

Probleme mit der Überprüfung von Software

Arten von Tests

Strategien für die Überprüfung von Software

Überprüfbarkeit und Zuverlässigkeit

Zentrale Voraussetzungen für wissenschaftlichen Erkenntnisgewinn

> *Testing is considered a core principle of scientific software because its impact is at the heart of knowledge generation.*
>
> – Scopatz und Huff

- Wissenschaftlichkeit erfordert Überprüfbarkeit.
  - unabhängige Überprüfung und ggf. Wiederholung
- Software ist oft integraler Bestandteil von Forschung.
  - Auswertungen sind in der Regel nichttrivial.
  - Eine Überprüfung auf Korrektheit ist ebenso nichttrivial.
- Wissenschaftliche Software muss zuverlässig sein.
  - Erkenntnisse und Fortschritt gründen auf Ergebnissen, die mithilfe von Software gewonnen werden.

Scopatz und Huff: Effective Computation in Physics, O'Reilly, Sebastopol 2015, S. 404

Wissenschaftliche Software ist selten getestet

Gründe sind mangelndes Bewusstsein und Wissen.

> *We would like to think that scientists are rigorous enough to realize the importance of testing [...]*
>
> *The truth of the matter is that most scientists are poorly equipped to truly test their code. The average blog or image-sharing website is better tested than most scientific software.*
>
> – Scopatz und Huff

- Gründe: mangelndes Bewusstsein und Wissen
  - Programmierung hat einen viel zu geringen Stellenwert.
  - Wissenschaftler sind selten ausgebildete Programmierer.

Scopatz und Huff: Effective Computation in Physics, O'Reilly, Sebastopol 2015, S. 405

Konsequenz: mangelnde Wissenschaftlichkeit

Wissenschaftliche Software ist meist nicht vertrauenswürdig.

> *If experimentalists don't calibrate their equipment,*
> *check their reagants' purity, and take careful notes,*
> *what they're doing isn't considered science.*
> *In contrast, computationalists don't even learn*
> *how to assess their software's quality in any systematic way,*
> *and very few would be able to recreate and rerun*
> *the programs they used to produce last year's papers.*
> *As a result, most computational science*
> *is irreproducible and of unknown quality.*
>
> – Greg Wilson

- ☛ Sorgfältige Tests wissenschaftlicher Software sind eine Frage der *Professionalität als Wissenschaftler*!

Wilson, *Comput. Sci. Eng.* **8**:66, 2006

Übersicht

Grundlegende Probleme von Tests

Eine Übersicht

- Tests sind destruktiv.
  - Ein erfolgreicher Test ist ein Test, der zu einem Fehler führt.
  - widerspricht dem Entwicklergedanken: etwas erschaffen
- Tests erfordern die Erwartung, Fehler zu finden.
  - Wer keine Fehler finden will, wird sie oft übersehen.
  - Entwicklern fehlt häufig die notwendige Objektivität.
- Tests sagen nichts über Fehlerfreiheit.
  - Tests können nur (bislang unbekannte) Fehler finden.
  - Die Abwesenheit von Fehlern ist allgemein nicht beweisbar.
- Tests selbst erhöhen nicht die Softwarequalität.
  - Testergebnisse müssen ernst genommen und Fehlerursachen behoben werden.

Praktische Probleme mit Tests

Eine Spurensuche, warum so wenig getestet wird

- Programme sind meist schwer zu testen.
  - Testbarkeit erfordert Modularität und Unabhängigkeit.
  - Entwicklung fokussiert meist nicht auf Testbarkeit.
- Tests haben einen viel zu geringen Stellenwert.
  - Tests fallen oft dem Zeitmangel zum Opfer.
  - Das Bewusstsein für die Bedeutung verlässlicher Software gerade in der Wissenschaft fehlt überwiegend.
- Es fehlt das Wissen für effektive Tests.
  - Tests zu schreiben ist nicht trivial.
- Testen ist langweilig und fehleranfällig.
  - Automatisierung ist das Gebot der Stunde.
  - Tests sollten automatisiert *ausgewertet* werden.

Vollständige Tests sind meist undurchführbar

Entscheidend ist die geschickte Wahl aussagekräftiger Testfälle.

Satz

Der mögliche Parameterraum jedes nichttrivialen Programms ist zu groß für vollständige Tests.

- Beispiel: Funktion zum Speichern von Adressen
  - Name/Adresse: 20 Buchstaben, Telefonnummer: 10 Ziffern
  - Möglichkeiten: $26^{20} \times 26^{20} \times 10^{10} \approx 10^{66}$
  - Vergleich: vermutlich ca. 10^{80} Atome im Universum
- Abhilfe: intelligente Wahl der Testfälle
  - charakteristische Testfälle
  - erfordert Vertrautheit mit der Problemstellung und Kenntnis über Spezialfälle

Vollständige Tests sind meist undurchführbar

Es gibt kein Kriterium für ausreichend viele Tests

> *Once you are confident that your tests are correct,*
> *and are finding bugs you create,*
> *how do you know if you have tested the code base*
> *thoroughly enough?*
>
> *The short answer is "you don't," and you never will.*
>
> – Andrew Hunt und David Thomas

- Tests zu schreiben ist eine Daueraufgabe
  - reflektiert das zunehmende Verständnis der Problemstellung
- Statistik: Es gibt mehr Nutzer als Entwickler.
  - Entwickler werden nie alle Fehler finden.
  - systematischer Umgang mit Fehlerberichten

Hunt und Thomas: The Pragmatic Programmer, Addison Wesley, Boston 1999, S. 245

Fehlerfreiheit ist eine Illusion

Systematische Tests können das notwendige Vertrauen schaffen.

Satz

Fehlerfreiheit von Programmen ist im Allgemeinen
weder erreichbar noch beweisbar.

- Fehlerfreiheit ist kein sinnvolles Kriterium.
  - Aber: Unerreichbarkeit ist kein Argument gegen Tests.
- Vertrauenswürdigkeit von Code ist entscheidend.
  - Systematische Tests können Vertrauen schaffen.
  - Entscheidend ist nicht die Zahl von Tests,
    sondern ihre Abdeckung unterschiedlicher Fälle.
  - Abdeckung aller Zustände ist entscheidend.
  - Abdeckung aller Codezeilen ist ein schlechtes Maß.

Komplexe Rechnungen sind schwer überprüfbar

Abhilfe: Test der Einzelteile und unabhängige Implementierungen

These

Numerische Simulationen und Rechnungen lassen sich oft nicht anhand bekannter Lösungen überprüfen.

- Erkenntnisgewinn ist Ziel der Wissenschaft.
  - Oft gibt es keine bekannten Lösungen.
  - Software ist oft Werkzeug des Erkenntnisgewinns.
- Lösungsansatz: Vertrauen in die Einzelteile
  - Elementaroperationen müssen zuverlässig korrekt sein.
  - Voraussetzung: modularer, lesbarer, testbarer Code
- Lösungsansatz: analytische Lösungen für Spezialfälle
  - erfordert tiefgreifendes Verständnis der zugrundeliegenden Theorie
- Lösungsansatz: unabhängige Implementierung
  - Implementierungsfehler sollten statistisch unabhängig sein.

Übersicht

Kriterien zur Einteilung von Tests

Tests lassen sich nach diversen Kriterien unterscheiden.

- Durchführende
  - Entwickler, Tester, Kunden, Anwender
- Größe der zu testenden Einheit
  - Unittests, Integrationstests, Systemtests
- Kenntnis der Implementierung
  - White-Box-Tests, Black-Box-Tests
- Reproduzierbarkeit
  - reproduzierbar, abhängig von äußeren Faktoren
- Art der Durchführung
  - manuell, automatisiert
- Zielstellung
  - Regression, Validierung, Entwicklung

- Entwickler sind für die Tests verantwortlich.
  - Entwickler sind in vielen Fällen selbst Nutzer.
  - Entwickler sind meist Einzelkämpfer.
  - Implementierung ist in der Regel bekannt.
- Wissenschaftliche Software dient dem Erkenntnisgewinn.
  - korrekte Lösungen für komplexe Probleme nicht bekannt
  - Ausweg: Überprüfung der einzelnen Codebestandteile
  - Ausweg: Analytische Lösungen für Grenzfälle und Abschätzen von Trends bei systematischer Variation einzelner Parameter
- Tests erfordern Kompetenzen auf zwei Gebieten.
  - Gebiete: Programmierung und Wissenschaft
  - Implementierungen sind immer fehleranfällig.
  - Korrekte Implementierungen garantieren nicht wissenschaftliche Korrektheit.

Terminologie: relevante Arten von Tests

Tests mit Relevanz im wissenschaftlichen Kontext

- Unittest
  - Test eines Codeblocks in Isolation
  - Getestete Codeblöcke sind in der Regel klein (meist eine Funktion oder Methode).
  - Verhalten des Codeblocks ist eindeutig definierbar.
- Integrationstest
  - Test des Zusammenspiels mehrerer Codeblöcke
  - Einzelne Codeblöcke wurden vorher getestet (Unittests).
  - Test nicht mehr (notwendigerweise) in Isolation
- Regressionstest
  - Test auf Übereinstimmung mit früherer Funktionalität
  - relevant bei Veränderungen im Code
  - stellt sicher, dass korrekter Code korrekt bleibt
  - Voraussetzung: Korrektheit der Tests

Übersicht

Allgemeine Regeln

Software sollte früh, oft und automatisiert getestet werden.

> *The single most important rule of testing is to do it.*
>
> – Kernighan und Pike

- früh testen
  - erleichtert das Eingrenzen von Fehlern
  - verhindert Fehlentwicklungen
- oft testen
  - mindestens einmal pro (Arbeits-)Tag
  - Fehler beheben, bevor weiter entwickelt wird
- automatisiert testen
  - Voraussetzung für häufiges Testen
  - verringert die Fehlerquote beim Testen

Kernighan und Pike: The Practice of Programming, Addison Wesley, Boston 1999, S. 162

Strategien für konkrete Tests

Eine erste Übersicht

- systematische Tests
  - jede Zeile im Produktivcode abtesten
  - jede Bedingung einzeln abtesten
  - Ziel: alle möglichen Zweige des Programms abtesten
- Kontrollfluss und Datenfluss testen
  - drei Zustände für Daten: definiert, benutzt, freigegeben
  - Atypische Folgen von Zuständen deuten auf Fehler hin.
  - bei dynamisch typisierten Sprachen sehr viel impliziter
- Grenzen analysieren
  - Grenzen von Wertebereichen sind besonders fehleranfällig.
  - besondere Vorsicht bei mehreren abhängigen Parametern
- Hilfsmittel zum Test in Isolation verwenden
  - Ziel: Test eines einzelnen Codeblocks
  - externe Abhängigkeiten durch Testumgebung auflösen

Wie viele Tests sind ausreichend?

Die McCabe-Metrik liefert die minimale Zahl von Testfällen.

Die McCabe-Metrik als untere Grenze

- Ziel
  - vollständige Abdeckung der Zweige eines Programms mit der minimalen Anzahl von Tests.
- Regeln zur Bestimmung der Zahl notwendiger Tests
  - 1 – gerader Weg durch das Programm
  - +1 für jedes `if`, `while`, `repeat`, `for`, `and`, `or`
  - +1 für jedes `case`
  - +1 wenn kein default-Zweig im `switch/case`
- Grenzen
  - deckt lediglich alle Zweige eines Programms ab
  - Überprüfung der Wertebereiche von Parametern erfordert zusätzliche Tests.

Thomas J. McCabe, *IEEE Trans. Softw. Eng.* 2:308, 1976

Tests selbst sind nicht fehlerfrei

Überprüfung der Tests führte zu einer unendlichen Regression.

Problem

- Tests sind Code und deshalb genauso fehleranfällig.
  - Test der Tests ist keine Lösung.

mögliche Lösungsstrategien

- Testcode ernst nehmen
  - Testcode ist genauso wichtig wie Produktivcode.
  - Testcode sollte genauso sorgfältig geschrieben werden.
  - Alle genannten Aspekte von „Sauberem Code“ anwenden.
- Testfälle frühzeitig planen
  - hilfreich zur Vermeidung falscher Annahmen
- Testcode nicht wegwerfen
  - hilft dabei, Testcode ernst zu nehmen
- Tests *vor* dem Produktivcode schreiben

Umgang mit Testcode

Testcode ist genauso wichtig wie Produktivcode.

- Testcode nicht wegwerfen
  - Tests sind ein wichtiger Bestandteil der Entwicklung.
  - Tests sollten immer wieder ausgeführt werden können.
- Testcode sauber schreiben
  - Lesbarkeit ist mindestens so wichtig wie bei Produktivcode.
  - Tests sind ausführbare Spezifikation und Dokumentation.
- Testcode zentral lagern
  - eigenes Verzeichnis oder direkt neben dem Produktivcode
  - Konsistente Benennung erhöht die Übersichtlichkeit.
- Testcode von Produktivcode trennen
  - Testcode dient *nicht* dem Debugging.
  - Produktivcode sollte *von außen* testbar sein
    (Test der Schnittstellen, nicht der Implementierung).

Regeln zur Überprüfung von Software

1. Testen
2. Systematisch testen
3. Testcode nicht wegwerfen
   und genauso ernst nehmen wie Produktivcode
4. Tests automatisieren

Parallele zum Vorgehen in den Wissenschaften

1. Überprüfung von Hypothesen
2. Systematisches Vorgehen
3. Dokumentation aller Schritte
4. Automatisierung von Teilaspekten

Zentrale Aspekte

- Überprüfung ist ein Kernaspekt sowohl von Wissenschaft als auch von moderner Softwareentwicklung.
- Überprüfung wissenschaftlicher Software ist entscheidend, weil sie den Erkenntnisgewinn selbst betrifft.
- Der größte Teil wissenschaftlicher Software ist unzureichend oder gar nicht getestet.
- Tests können nur die Existenz von Fehlern beweisen, niemals deren Abwesenheit.
- Tests sollten so früh wie möglich, so oft wie möglich und so automatisiert wie möglich durchgeführt werden.