Einblicke ins Testen und die testgetriebene Entwicklung (TDD) in wissenschaftlicher Softwareentwicklung

Im Seminar „Softwareentwicklung in der Wissenschaft“ 2015

Hamburg 5. Mai 2015

Marek.Jacob@studium.uni-hamburg.de

Ausgangspunkte

- Nanthaamornphong et al., Mai-Juni 2014
  "*Building CLiiME via Test-Driven Development: A Case Study*" – *Neuentwicklung*
- Clune, T.L. und Rood, R.B., Nov.-Dez 2011
  "*Software Testing and Verification in Climate Model Development*" – *Rewrite & Neuentwicklung*
- Rilee, M.; Clune, T., 2014
  "*Towards Test Driven Development for Computational Science with pFUnit*" – *Frameworks*

Agenda

- Bestandsaufnahme
- Voraussetzungen für TDD
- Beispiele
- Fazits

Agenda

- Bestandsaufnahme
- Voraussetzungen für TDD
- Beispiele
- Fazits

Ziele und Begrifflichkeit

- Ziel der wissenschaftlichen Softwareentwicklung
  - Das wissenschaftliche Verständnis erhöhen

- Vergleiche
  - *Validation*: mit Beobachtungen / Messungen
  - *Verifikation*: mit analytischen Testfällen und anders berechneten Ergebnissen

Haltung gegenüber Tests

- Modellvalidation und Softwareverifikation werden vermischt

  [...] the community isn’t aware of pervasive unit testing’s benefits and has misconceptions about the nature and difficulty of implementing unit tests.

  [Clune, 2011, „Software Testing and Verification in Climate Model Development“]

- eigener wissenschaftlicher Teststil in Forschergruppen

„Bildet die Software die Realität ab?“

- Tests auf Systemebene z.B.
  - Realistische Eingangsdaten
  - Stürzt die Simulation ab?
  - paralleler Betrieb
  - Checkpoints
- Übereinstimmung des theoretischen Modells und der Software wird angenommen
- Komponententests / Unittests
  - Filigrane Tests meist nicht vorhanden

Agenda

- Bestandsaufnahme
- Voraussetzungen für TDD
- Beispiele
- Fazits

Warum braucht die Wissenschaft TDD?

- passt besser als *plan-driven development*
- reduziert Entwicklungsrisiken für Änderungen
- früh gefundene Fehler sind günstiger
- filigrane Tests laufen schneller
- motivierende Fortschrittsanzeige

- Erstrebenswert:
  - Testframework
  - Unittests ständig während der Programmierung, Systemtests nächtlich oder wöchentlich

Besonderheiten für wissenschaftliche Softwareentwicklung und Hochleistungsrechnen

- parallele Ausführung
- numerische Algorithmen
  - feinkörnig implementieren
  - Objektorientierung verwenden
- Fließkomma Arithmetik
  - Tests mit relativer und absoluter Toleranz
  - Filigrane Tests um Fehlerakkumulation zu verhindern
- (Chaostheorie)

Tests in der Wissenschaft

- Konflikt mit Genauigkeit numerischer Approximation
  - besser: synthetische Werte auf kleinen Gittern

    ⇒ Überschaubares Ergebnis
- Altlastencode / Legacy code
  - wenn möglich in testbare Teile zerlegen
  - Ansatz: neue Funktionalität in testbare Funktionen diese aus Altlast nur aufrufen.
  - Bei Erweiterung Beständigkeit der Altlast testen

Agenda

- Bestandsaufnahme
- Voraussetzungen für TDD
- Beispiele
- Fazits

PARAMESH – Tests vor TDD

- ist ein Framework zur parallelen Implementation von mehrdimensionalen partiellen Differentialgleichungen
- u.a. automatische Gitterverfeinerung

Tests

- während der Entwicklung sehr wichtig
- keine Unittests, Teststframework oder Einheitlichkeit
- Standalone Programme, meist per Hand ausgeführt
- „eigener“ Codeabschnitt, „eigene“ Tests
- Tests nicht dokumentiert

TDD Bsp.: Community Laser-Induced Incandescence Modeling Environment – CLiiME

- Simuliert von Laser angeregtes Glühen von Partikeln
- erstes agiles Projekt am Institut
- 3 Entwickler und 1 Fachexperte
- Fortran für die Numerik, Java fürs GUI

Tests

- echte *Unittests* mit Testklassen
- kurze Tests ⇒ übersichtlich, präzise

example of unit testing code for the absorption energy.

```
subroutine absorption_energy_input(this)
    [...]
    real(rkind) :: expect = 2.441e-10
    absorption = absorptionEnergy()
    Qabs = absorption%Energy()
    call assert(error_within_tolerance(expect, Qabs), &
        error_message("Qabs = 2.441e-10 is expected"))
end subroutine

subroutine absorption_energy_default(this)
    [...]
    !Result from running the Igor code
    real(rkind) :: expect = 1.441E-10
    character(len=*) :: filename='default.txt'
    absorption = absorptionEnergy(filename=filename)
    Qabs = absorption%Energy(properties,laser)
    call assert(error_within_tolerance(expect, Qabs), &
        error_message("Qabs = 1.44E-10 is expected"))
end subroutine
```

[Nanthaamornphong, 2014, Figure 2]

Numerische Funktionen in CLiiME

- Zerlegung in einfache Operationen
- Test: Vergleich mit extern berechneten Beispielwerten
- relative Fehlertoleranz von 10 %

TDD Regeln in CLiiME

- Unabhängigkeit
- keine manuelles Setup
- Test sollen schnell laufen
- Entwickeln bis Tests erfüllt werden
- Test erstellen, wenn ein Bug gefunden wird
- Tests müssen vor Repository Check-in erfüllt sein

Weitere TDD Beispiele

- *Gtraj* – Rewrite
  - 2 Fehler gefunden:
    Grenzfallproblem, Fehler in komplexer Numerik
  - Wissenschaftler wollten erst implementieren, dann testen
  - Vernachlässigung bei Erfolgsphase
- *Snowflake* – Neuentwicklung
  - saubere Mathematik als Vorlage
  - 12h Programmierung
    ⇒ 800 Zeilen Tests, 700 Zeilen Implementation
  - auf Anhieb korrekt

Agenda

- Bestandsaufnahme
- Voraussetzungen für TDD
- Beispiele
- Fazits

Fazits I

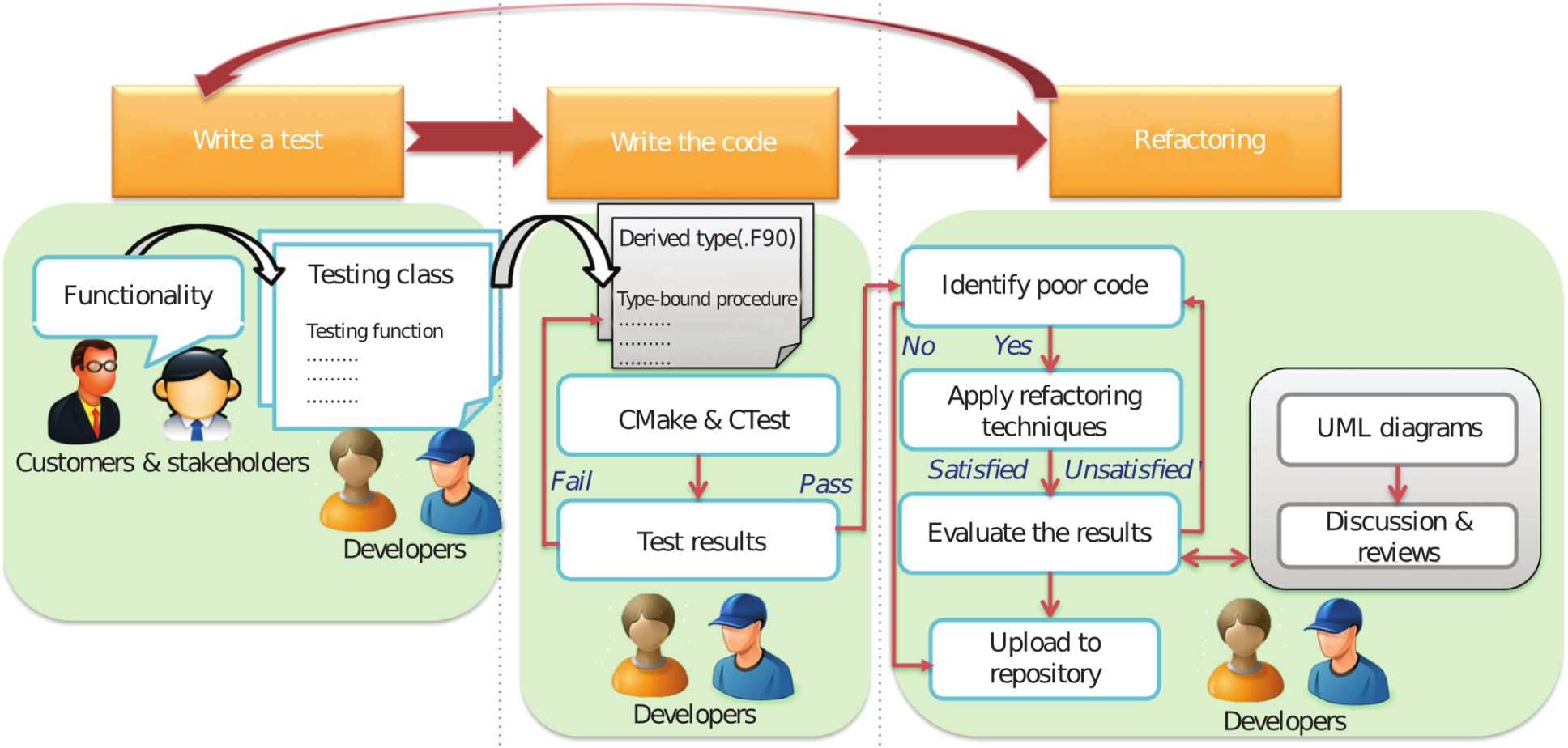
- Tests…
  - fördern Wartbarkeit und flexible Software
  - dokumentieren Sinn und Zweck gewisser Codeteile
  - ermöglichen schadenloses Refactoring

    ⇒ bessere Codequalität
  - ermöglichen einfachere Erweiterung
  - definieren konkrete Ansprüche
  - dienen als Anwendungsbeispiel
  - bewirken Erfolgserlebnisse
  - reduzieren explizite Autorenzuordnung
- Filigrane Tests…
  - lassen sich schnell und ständig ausführen

Fazits II Wissenschaft sensibilisieren

- Implementation ≠ Theorie
- Tests fördern Verständnis der Theorie
- Tests zeigen, dass das wissenschaftliche Verständnis vertrauenswürdig umgesetzt ist
  ⇒ Vertrauen ins Modell steigt
  ⇒ Vertrauen in Prognosefähigkeit steigt
- Zusammenspiel Wissenschaft und Softwareentwickler
  - TDD verbessert Zusammenarbeit
    - Wissenschaftler stellt Anforderungen
    - Entwickler entwickelt und verbessert Code

Wissenschaftler Entwickler



[Nanthaamornphong, 2014, Figure 1]

Fazit III Probleme

- Altlasten
  - Benötigen besondere Beachtung
- striktes TDD liefert keine Antwort für numerische Funktionen
- Filigrane Tests mit Objekten und Mocks in Tests
  - Mocks sind (in Fortran) kaum automatisch zu generieren
  - Objektorientierung müsste sich erst durchsetzen
- es werden automatische easy-to-use und easy-to-lern Frameworks gebraucht
- Unterstützung durch IDE

Einschub: pFUnit (seit 2005)

Paralleles Fortran Unittesting Framework

- Für
  - Mehrdimensionale Arrays
  - Fließkommazahlen
    - Relative und absolute Fehler, NaNs
  - Parallelismus
    - MPI
    - OpenMP
- Mockgenerator

Fazit IV Ratschläge

- Vor Projektstart: Schulung in agiler Softwareentwicklung und TDD
- vorteilhaft: Mentor für Agilität
- bei „größeren“ Projekten: zentrale Fehlerverwaltung
- Evaluation der Tests
- Beachte: Mehr Code zu warten

Mein Fazit

- Testgetriebene Entwicklung kann in Wissenschaft angewendet werden
- TDD benötigt Lernprozess bei Wissenschaftlern
- Viel Hoffnung in TDD
- Mehr Softwareentwickler in Wissenschaft
  ⇒ besserer Wissenschaftlicher Code
- TDD ist ein guter aber anspruchsvoller Ansatz

Quellen

- Nanthaamornphong, A.; Carver, J.C.; Morris, K.; Michelsen, H.A.; Rouson, D.W.I.,
  "Building CLiiME via Test-Driven Development: A Case Study", Computing in Science & Engineering , vol.16, no.3, pp.36,46, May-June 2014, doi: 10.1109/MCSE.2014.33
- Clune, T.L.; Rood, R.B.,
  "Software Testing and Verification in Climate Model Development", Software, IEEE , vol.28, no.6, pp.49,55, Nov.-Dec. 2011, doi: 10.1109/MS.2011.117
- Rilee, M.; Clune, T.,
  "Towards Test Driven Development for Computational Science with pFUnit", Software Engineering for High Performance Computing in Computational Science and Engineering (SE-HPCCSE), 2014 Second International Workshop on , vol., no., pp.20,27, 21-21 Nov. 2014, doi: 10.1109/SE-HPCCSE.2014.5